CHAPTER 1

REVIEW OF LOGIC DESIGN FUNDAMENTALS

This chapter reviews many of the logic design topics normally taught in a first course in logic design. Some of the review examples that follow are referenced in later chapters of this text. For more details on any of the topics discussed in this chapter, the reader should refer to a standard logic design textbook such as *Fundamentals of Logic Design*, 4th ed. (Boston: PWS Publishing Company, 1995).

First, we review combinational logic and then sequential logic. Combinational logic has no memory, so the present output depends only on the present input. Sequential logic has memory, so the present output depends not only on the present input but also on the past sequence of inputs. The sections on sequential network timing and synchronous design are particularly important, since a good understanding of timing issues is essential to the successful design of digital systems.

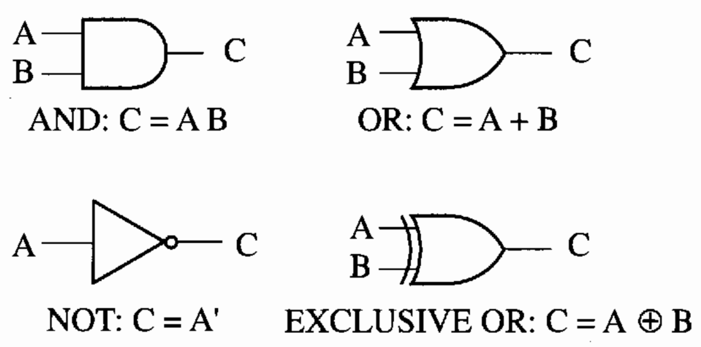
1.1 COMBINATIONAL LOGIC

Some of the basic gates used in logic networks are shown in Figure 1-1. Unless otherwise specified, all the variables that we use to represent logic signals will be two-valued, and the two values will be designated 0 and 1. We will normally use positive logic, for which a low voltage corresponds to a logic 0 and a high voltage corresponds to a logic 1. When negative logic is used, a low voltage corresponds to a logic 1 and a high voltage corresponds to a logic 0.

For the AND gate of Figure 1-1, the output $C = 1$ if and only if the input $A = 1$ *and* the input $B = 1$. We will use a raised dot or simply write the variables side by side to indicate the AND operation; thus $C = A \cdot B = AB$. For the OR gate, the output $C = 1$ if and only if the input $A = 1$ *or* the input $B = 1$ (inclusive OR). We will use + to indicate the OR operation; thus $C = A + B$. The NOT gate, or inverter, forms the complement of the input; that is, if $A = 1$, $C = 0$, and if $A = 0$, $C = 1$. We will use a prime (') to indicate the complement (NOT) operation, so $C = A'$. The exclusive-OR (XOR) gate has an output $C = 1$ if $A = 1$ and $B = 0$ or if $A = 0$ and $B = 1$. The symbol ⊕ represents exclusive OR, so we write

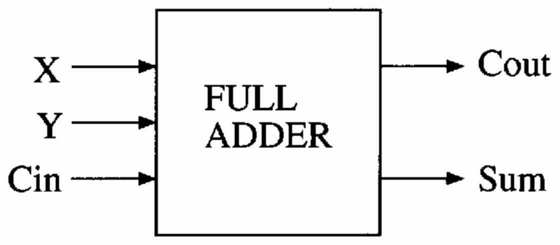
$$C = AB' + A'B = A \oplus B \tag{1-1}$$

Figure 1-1 Basic Gates



The behavior of a combinational logic network can be specified by a truth table that gives the network outputs for each combination of input values. As an example, consider the full adder of Figure 1-2, which adds two binary digits (*X* and *Y*) and a carry (*Cin*) to give a sum (*Sum*) and a carry out (*Cout*). The truth table specifies the adder outputs as a function of the adder inputs. For example, when the inputs are $X = 0$, $Y = 0$ and $Cin = 1$, adding the three inputs gives $0 + 0 + 1 = 01$, so the sum is 1 and the carry out is 0. When the inputs are 011, $0 + 1 + 1 = 10$, so $Sum = 0$ and $Cout = 1$. When the inputs are $X = Y = Cin = 1$, $1 + 1 + 1 = 11$, so $Sum = 1$ and $Cout = 1$.

Figure 1-2 Full Adder



X	Y	Cin	Cout	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a) Full adder module **(b)** Truth table

We will derive algebraic expressions for *Sum* and *Cout* from the truth table. From the table, $Sum = 1$ when $X = 0$, $Y = 0$, and $Cin = 1$. The term $X'Y'Cin$ equals 1 only for this combination of inputs. The term $X'YCin' = 1$ only when $X = 0$, $Y = 1$, and $Cin = 0$. The term $XY'Cin'$ is 1 only for the input combination $X = 1$, $Y = 0$, and $Cin = 0$. The term $XYCin$ is 1 only when $X = Y = Cin = 1$. Therefore, *Sum* is formed by ORing these four terms together:

$$Sum = X'Y'Cin + X'YCin' + XY'Cin' + XYCin \quad \textbf{(1-2)}$$

Each of the terms in this expression is 1 for exactly one combination of input values. In a similar manner, *Cout* is formed by ORing four terms together:

$$Cout = X'YCin + XY'Cin + XYCin' + XYCin \qquad \textbf{(1-3)}$$

Each term in equations (1-2) and (1-3) is referred to as a *minterm*, and these equations are referred to as *minterm expansions*. These minterm expansions can also be written in *m*-notation or decimal notation as follows:

$$Sum = m_1 + m_2 + m_4 + m_7 = \Sigma m(1, 2, 4, 7)$$

$$Cout = m_3 + m_5 + m_6 + m_7 = \Sigma m(3, 5, 6, 7)$$

The decimal numbers designate the rows of the truth table for which the corresponding function is 1. Thus *Sum* = 1 in rows 001, 010, 100, and 111 (rows 1, 2, 4, 7).

A logic function can also be represented in terms of the inputs for which the function value is 0. Referring to the truth table for the full adder, $Cout = 0$ when $X = Y = Cin = 0$. The term $(X + Y + Cin)$ is 0 only for this combination of inputs. The term $(X + Y + Cin')$ is 0 only when $X = Y = 0$ and $Cin = 1$. The term $(X + Y' + Cin)$ is 0 only when $X = Cin = 0$ and $Y = 1$. The term $(X' + Y + Cin)$ is 0 only when $X = 1$ and $Y = Cin = 0$. *Cout* is formed by ANDing these four terms together:

$$Cout = (X + Y + Cin)(X + Y + Cin')(X + Y' + Cin)(X' + Y + Cin) \qquad \textbf{(1-4)}$$

Cout is 0 only for the 000, 001, 010, and 100 rows of the truth table and therefore must be 1 for the remaining four rows. Each of the terms in (1-4) is referred to as a *maxterm*, and (1-4) is called a *maxterm expansion*. This *maxterm expansion* can also be written in decimal notation as

$$Cout = M_0 \cdot M_1 \cdot M_2 \cdot M_4 = \Pi M(0, 1, 2, 4)$$

where the decimal numbers correspond to the truth table rows for which $Cout = 0$.

1.2 BOOLEAN ALGEBRA AND ALGEBRAIC SIMPLIFICATION

The basic mathematics used for logic design is Boolean algebra. Table 1-1 summarizes the laws and theorems of Boolean algebra. They are listed in dual pairs and can easily be verified for two-valued logic by using truth tables. These laws and theorems can be used to simplify logic functions so they can be realized with a reduced number of components.

DeMorgan's laws (1-16, 1-16D) can be used to form the complement of an expression on a step-by-step basis. The generalized form of DeMorgan's law (1-17) can be used to form the complement of a complex expression in one step. Equation (1-17) can be interpreted as follows: To form the complement of a Boolean expression, replace each variable by its complement; also replace 1 with 0, 0 with 1, OR with AND, and AND with OR. Add parentheses as required to assure the proper hierarchy of operations. If AND is performed before OR in F, then parentheses may be required to assure that OR is performed before AND in F'.

Example

$$F = X + E'K\,(\,C\,(AB + D') \cdot 1 + WZ'\,(G'H + 0)\,)$$

$$F' = X'\,(\,E + K' + (C' + (A' + B')\,D + 0)\,(W' + Z + (G + H') \cdot 1\,)\,)$$

The boldface parentheses in F' were added when an AND operation in F was replaced with an OR. The dual of an expression is the same as its complement, except that the variables are not complemented.

Table 1-1 Laws and Theorems of Boolean Algebra

Operations with 0 and 1:			
$X + 0 = X$	(1-5)	$X \cdot 1 = X$	(1-5D)
$X + 1 = 1$	(1-6)	$X \cdot 0 = 0$	(1-6D)
Idempotent laws:			
$X + X = X$	(1-7)	$X \cdot X = X$	(1-7D)
Involution law:			
$(X')' = X$	(1-8)		
Laws of complementarity			
$X + X' = 1$	(1-9)	$X \cdot X' = 0$	(1-9D)
Commutative laws:			
$X + Y = Y + X$	(1-10)	$XY = YX$	(1-10D)
Associative laws:			
$(X + Y) + Z = X + (Y + Z) = X + Y + Z$	(1-11)	$(XY)Z = X(YZ) = XYZ$	(1-11D)
Distributive laws:			
$X(Y + Z) = XY + XZ$	(1-12)	$X + YZ = (X + Y)\,(X + Z)$	(1-12D)
Simplification theorems:			
$XY + XY' = X$	(1-13)	$(X + Y)\,(X + Y') = X$	(1-13D)
$X + XY = X$	(1-14)	$X(X + Y) = X$	(1-14D)
$(X + Y')Y = XY$	(1-15)	$XY' + Y = X + Y$	(1-15D)
DeMorgan's laws:			
$(X + Y + Z + \cdots)' = X'Y'Z'\cdots$	(1-16)	$(XYZ\cdots)' = X' + Y' + Z' + \cdots$	(1-16D)
$[f(X_1, X_2, \ldots, X_n, 0, 1, +, \cdot)]' = f(X_1', X_2', \ldots, X_n', 1, 0, \cdot, +)$			(1-17)
Duality:			
$(X + Y + Z + \cdots)^D = XYZ\cdots$	(1-18)	$(XYZ\cdots)^D = X + Y + Z + \cdots$	(1-18D)
$[f(X_1, X_2, \ldots, X_n, 0, 1, +, \cdot)]^D = f(X_1, X_2, \ldots, X_n, 1, 0, \cdot, +)$			(1-19)
Theorem for multiplying out and factoring:			
$(X + Y)(X' + Z) = XZ + X'Y$	(1-20)	$XY + X'Z = (X + Z)(X' + Y)$	(1-20D)
Consensus theorem:			
$XY + YZ + X'Z = XY + X'Z$	(1-21)	$(X + Y)(Y + Z)(X' + Z) = (X + Y)(X' + Z)$	(1-21D)

Four ways of simplifying a logic expression using the theorems in Table 1-1 are as follows:

1. *Combining terms.* Use the theorem $XY + XY' = X$ to combine two terms. For example,

$$ABC'D' + ABCD' = ABD' \quad [X = ABD',\ Y = C]$$

When combining terms by this theorem, the two terms to be combined should contain exactly the same variables, and exactly one of the variables should appear complemented in one term and not in the other. Since $X + X = X$, a given term may be duplicated and combined with two or more other terms. For example, the expression for *Cout* (Equation 1-3) can be simplified by combining the first and fourth terms, the second and fourth terms, and the third and fourth terms:

$$Cout = (X'YCin + XYCin) + (XY'Cin + XYCin) + (XYCin' + XYCin)$$

$$= YCin + XCin + XY \tag{1-22}$$

Note that the fourth term in (1-3) was used three times.

The theorem can still be used, of course, when X and Y are replaced with more complicated expressions. For example,

$$(A + BC)(D + E') + A'(B' + C')(D + E') = D + E'$$

$$[X = D + E',\ Y = A + BC,\ Y' = A'(B' + C')]$$

2. *Eliminating terms.* Use the theorem $X + XY = X$ to eliminate redundant terms if possible; then try to apply the consensus theorem ($XY + X'Z + YZ = XY + X'Z$) to eliminate any consensus terms. For example,

$$A'B + A'BC \quad = A'B \quad [X = A'B]$$

$$A'BC' + BCD + A'BD \quad = A'BC' + BCD \quad [X = C,\ Y = BD,\ Z = A'B]$$

3. *Eliminating literals.* Use the theorem $X + X'Y = X + Y$ to eliminate redundant literals. Simple factoring may be necessary before the theorem is applied. For example,

$$A'B + A'B'C'D' + ABCD' = A'(B + B'C'D') + ABCD' \quad \text{(by (1-12))}$$

$$= A'(B + C'D') + ABCD' \quad \text{(by (1-15D))}$$

$$= B(A' + ACD') + A'C'D' \quad \text{(by (1-10))}$$

$$= B(A' + CD') + A'C'D' \quad \text{(by (1-15D))}$$

$$= A'B + BCD' + A'C'D' \quad \text{(by (1-12))}$$

The expression obtained after applying 1, 2, and 3 will not necessarily have a minimum number of terms or a minimum number of literals. If it does not and no further simplification can be made using 1, 2, and 3, deliberate introduction of redundant terms may be necessary before further simplification can be made.

4. *Adding redundant terms*. Redundant terms can be introduced in several ways, such as adding XX', multiplying by $(X + X')$, adding YZ to $XY + X'Z$ (consensus theorem), or adding XY to X. When possible, the terms added should be chosen so that they will combine with or eliminate other terms. For example,

$WX + XY + X'Z' + WY'Z'$ (Add WZ' by the consensus theorem.)

$= WX + XY + X'Z' + WY'Z' + WZ'$ (eliminate $WY'Z'$)

$= WX + XY + X'Z' + WZ'$ (eliminate WZ')

$= WX + XY + X'Z'$

When multiplying out or factoring an expression, in addition to using the ordinary distributive law (1-12), the second distributive law (1-12D) and theorem (1-20) are particularly useful. The following is an example of multiplying out to convert from a product of sums to a sum of products:

$(A + B + D)(A + B' + C')(A' + B + D')(A' + B + C')$

$= (A + (B + D)(B' + C'))(A' + B + C'D')$ (by (1-12D))

$= (A + BC' + B'D)(A' + B + C'D')$ (by (1-20))

$= A(B + C'D') + A'(BC' + B'D)$ (by (1-20))

$= AB + AC'D' + A'BC' + A'B'D$ (by (1-12))

Note that the second distributive law and theorem (1-20) were applied before the ordinary distributive law. Any Boolean expression can be factored by using the two distributive laws and theorem (1-20D). As an example of factoring, read the steps in the preceding example in the reverse order.

The following theorems apply to exclusive-OR:

$X \oplus 0 = X$ **(1-23)**

$X \oplus 1 = X'$ **(1-24)**

$X \oplus X = 0$ **(1-25)**

$X \oplus X' = 1$ **(1-26)**

$$X \oplus Y = Y \oplus X \qquad \text{(commutative law)} \qquad \textbf{(1-27)}$$

$$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z) = X \oplus Y \oplus Z \qquad \text{(associative law)} \qquad \textbf{(1-28)}$$

$$X(Y \oplus Z) = XY \oplus XZ \qquad \text{(distributive law)} \qquad \textbf{(1-29)}$$

$$(X \oplus Y)' = X \oplus Y' = X' \oplus Y = XY + X'Y' \qquad \textbf{(1-30)}$$

The expression for *Sum* (equation (1-2)) can be rewritten in terms of exclusive-OR by using (1-1) and (1-30):

$$Sum = X'(Y'Cin + YCin') + X(Y'Cin' + YCin)$$

$$= X'(Y \oplus Cin) + X(Y \oplus Cin)' = X \oplus Y \oplus Cin \qquad \textbf{(1-31)}$$

1.3 KARNAUGH MAPS

Karnaugh maps provide a convenient way to simplify logic functions of three to five variables. Figure 1-3 shows a four-variable Karnaugh map. Each square in the map represents one of the 16 possible minterms of four variables. A 1 in a square indicates that the minterm is present in the function, and a 0 (or blank) indicates that the minterm is absent. An X in a square indicates that we don't care whether the minterm is present or not. Don't cares arise under two conditions: (1) The input combination corresponding to the don't care can never occur, and (2) the input combination can occur, but the network output is not specified for this input condition.

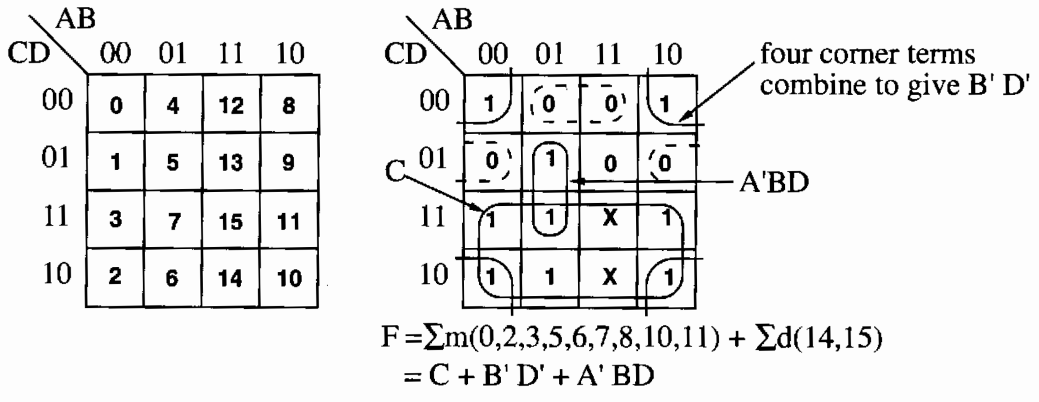
The variable values along the edge of the map are ordered so that adjacent squares on the map differ in only one variable. The first and last columns and the top and bottom rows of the map are considered to be adjacent. Two 1s in adjacent squares can be combined by eliminating one variable using $xy + xy' = x$. Figure 1-3 shows a four-variable function with nine minterms and two don't cares. Minterms $A'BC'D$ and $A'BCD$ differ only in the variable C, so they can be combined to form $A'BD$, as indicated by a loop on the map. Four 1s in a symmetrical pattern can be combined to eliminate two variables. The 1s in the four corners of the map can be combined as follows:

$$(A'B'C'D' + AB'C'D') + (A'B'CD' + AB'CD') = B'C'D' + B'CD' = B'D'$$

as indicated by the loop. Similarly, the six 1s and two Xs in the bottom half of the map combine to eliminate three variables and form the term C. The resulting simplified function is

$$F = A'BD + B'D' + C$$

Figure 1-3 Four-Variable Karnaugh Maps



F =Σm(0,2,3,5,6,7,8,10,11) + Σd(14,15)
= C + B' D' + A' BD

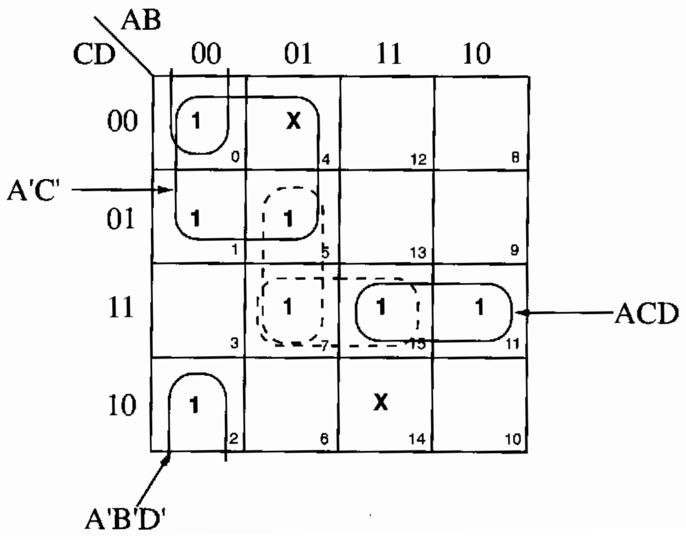
(a) Location of minterms **(b)** Looping terms

The minimum sum-of-products representation of a function consists of a sum of prime implicants. A group of one, two, four, or eight adjacent 1s on a map represents a prime implicant if it cannot be combined with another group of 1s to eliminate a variable. A prime implicant is essential if it contains a 1 that is not contained in any other prime implicant. When finding a minimum sum of products from a map, essential prime implicants should be looped first, and then a minimum number of prime implicants to cover the remaining 1s should be looped. The Karnaugh map shown in Figure 1-4 has three essential prime implicants. *A'C'* is essential because minterm m_1 is not covered by any other prime implicant. Similarly, *ACD* is essential because of m_{11}, and *A'B'D'* is essential because of m_2. After looping the essential prime implicants, all 1s are covered except m_7. Since m_7 can be covered by either prime implicant *A'BD* or *BCD*, *F* has two minimum forms:

$$F = A'C' + A'B'D' + ACD + A'BD$$

and $$F = A'C' + A'B'D' + ACD + BCD$$

Figure 1-4 Selection of Prime Implicants



When don't cares (Xs) are present on the map, the don't cares are treated like 1s when forming prime implicants, but the Xs are ignored when finding a minimum set of prime implicants to cover all the 1s. The following procedure can be used to obtain a minimum sum of products from a Karnaugh map:

1. Choose a minterm (a 1) that has not yet been covered.
2. Find all 1s and Xs adjacent to that minterm. (Check the n adjacent squares on an n-variable map.)
3. If a single term covers the minterm and all the adjacent 1s and Xs, then that term is an essential prime implicant, so select that term. (Note that don't cares are treated like 1s in steps 2 and 3 but not in step 1.)
4. Repeat steps 1, 2, and 3 until all essential prime implicants have been chosen.
5. Find a minimum set of prime implicants that cover the remaining 1s on the map. (If there is more than one such set, choose a set with a minimum number of literals.)

To find minimum product of sums from a Karnaugh map, loop the 0s instead of the 1s. Since the 0s of F are the 1s of F', looping the 0s in the proper way gives the minimum sum of products for F', and the complement is the minimum product of sums for F. For Figure 1-3, we can first loop the essential prime implicants of F' ($BC'D'$ and $B'C'D$, indicated by dashed loops), and then cover the remaining 0 with ABC' or $AC'D$. Thus one minimum sum for F' is

$$F' = BC'D' + B'C'D + ABC'$$

from which the minimum product of sums for F is

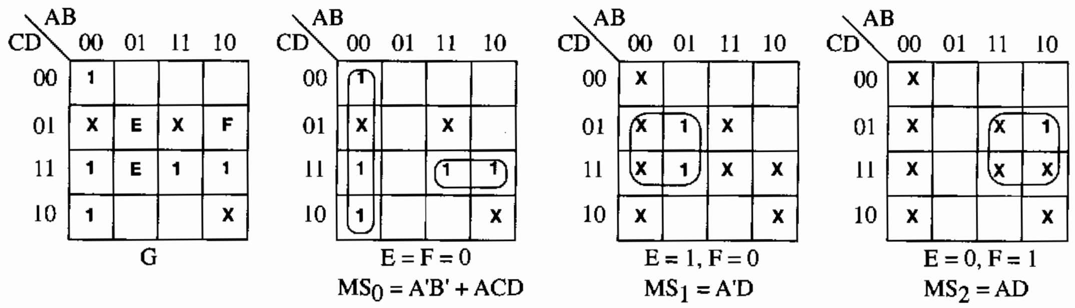
$$F = (B' + C + D)(B + C + D')(A' + B' + C)$$

By using map-entered variables, Karnaugh map techniques can be extended to simplify functions with more than four or five variables. Figure 1-5 shows a 4-variable map with two additional variables entered in the squares in the map. When E appears in a square, this means that if $E = 1$, the corresponding minterm is present in the function G, and if $E = 0$, the minterm is absent. Thus, the map represents the 6-variable function

$$G(A, B, C, D, E, F) = m_0 + m_2 + m_3 + Em_5 + Em_7 + Fm_9 + m_{11} + m_{15} \text{ (+ don't care terms)}$$

where the minterms are minterms of the variables A, B, C, D. Note that m_9 is present in G only when $F = 1$.

Figure 1-5 Simplification Using Map-Entered Variables



Next we will discuss a general method of simplifying functions using map-entered variables. In general, if a variable P_i is placed in square m_j of a map of function F, this means that $F = 1$ when $P_i = 1$ and the variables are chosen so that $m_j = 1$. Given a map with variables $P_1, P_2, \ldots$ entered into some of the squares, the minimum sum-of-products form of F can be found as follows: Find a sum-of-products expression for F of the form

$$F = MS_0 + P_1MS_1 + P_2MS_2 + \cdots \tag{1-32}$$

where

- MS_0 is the minimum sum obtained by setting $P_1 = P_2 = \cdots = 0$.
- MS_1 is the minimum sum obtained by setting $P_1 = 1$, $P_j = 0$ ($j \neq 1$), and replacing all 1s on the map with don't cares.
- MS_2 is the minimum sum obtained by setting $P_2 = 1$, $P_j = 0$ ($j \neq 2$), and replacing all 1s on the map with don't cares.

Corresponding minimum sums can be found in a similar way for any remaining map-entered variables.

The resulting expression for F will always be a correct representation of F. This expression will be a minimum provided that the values of the map-entered variables can be assigned independently. On the other hand, the expression will not generally be a minimum if the variables are not independent (for example, if $P_1 = P_2'$).

For the example of Figure 1-5, maps for finding MS_0, MS_1, and MS_2 are shown, where E corresponds to P_1 and F corresponds to P_2. The resulting expression is a minimum sum of products for G:

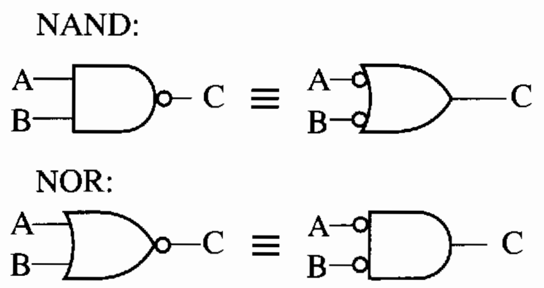
$$G = A'B' + ACD + EA'D + FAD$$

After some practice, it should be possible to write the minimum expression directly from the original map without first plotting individual maps for each of the minimum sums.

1.4 DESIGNING WITH NAND AND NOR GATES

In many technologies, implementation of NAND gates or NOR gates is easier than that of AND and OR gates. Figure 1-6 shows the symbols used for NAND and NOR gates. The *bubble* at a gate input or output indicates a complement. Any logic function can be realized using only NAND gates or only NOR gates.

Figure 1-6 NAND and NOR Gates



C = (AB)' = A' + B'

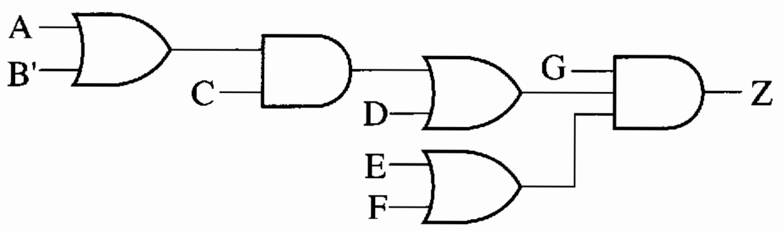
C = (A+B)' = A'B'

Conversion from networks of OR and AND gates to networks of all NOR gates or all NAND gates is straightforward. To design a network of NOR gates, start with a product-of-sums representation of the function (circle 0s on the Karnaugh map). Then find a network of OR and AND gates that has an AND gate at the output. If an AND gate output does not drive an AND gate input and an OR gate output does not connect to an OR gate input, then conversion is accomplished by replacing all gates with NOR gates and complementing inputs if necessary. Figure 1-7 illustrates the conversion procedure for

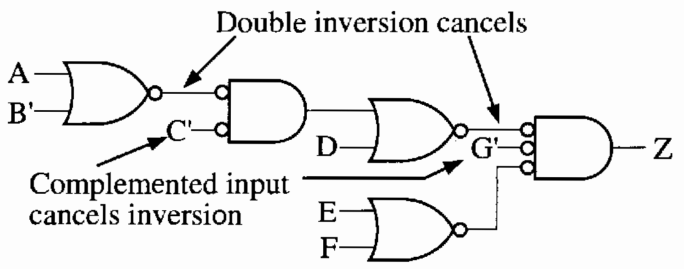
$$Z = G(E + F)(A + B' + D)(C + D) = G(E + F)[(A + B')C + D]$$

Conversion to a network of NAND gates is similar, except the starting point should be a sum-of-products form for the function (circle 1s on the map), and the output gate of the AND-OR network should be an OR gate.

Figure 1-7 Conversion to NOR Gates



(a) AND-OR network



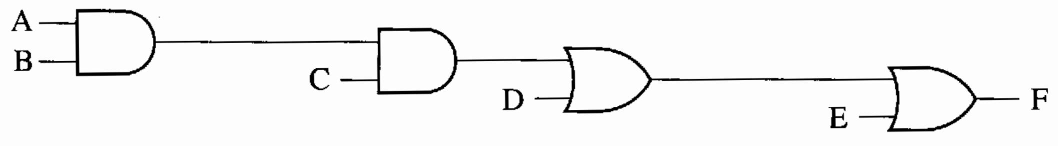
(b) Equivalent NOR-gate network

Even if AND and OR gates do not alternate, we can still convert a network of AND and OR gates to a NAND or NOR network, but it may be necessary to add extra inverters so that each added inversion is canceled by another inversion. The following procedure may be used to convert to a NAND (or NOR) network:

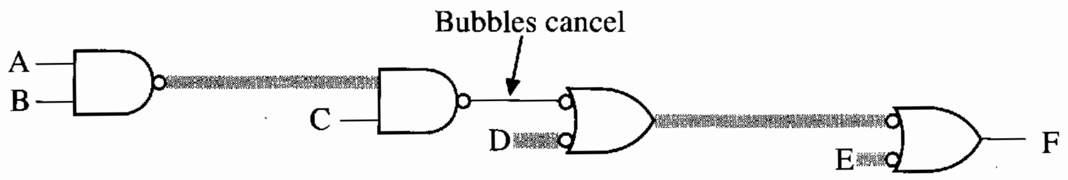
1. Convert all AND gates to NAND gates by adding an inversion bubble at the output. Convert OR gates to NAND gates by adding inversion bubbles at the inputs. (To convert to NOR, add inversion bubbles at all OR gate outputs and all AND gate inputs.)
2. Whenever an inverted output drives an inverted input, no further action is needed, since the two inversions cancel.
3. Whenever a non-inverted gate output drives an inverted gate input or vice versa, insert an inverter so that the bubbles will cancel. (Choose an inverter with the bubble at the input or output, as required.)
4. Whenever a variable drives an inverted input, complement the variable (or add an inverter) so the complementation cancels the inversion at the input.

In other words, if we always add bubbles (or inversions) in pairs, the function realized by the network will be unchanged. To illustrate the procedure, we will convert Figure 1-8(a) to NANDs. First, we add bubbles to change all gates to NAND gates (Figure 1-8(b)). The highlighted lines indicate four places where we have added only a single inversion. This is corrected in Figure 1-8(c) by adding two inverters and complementing two variables.

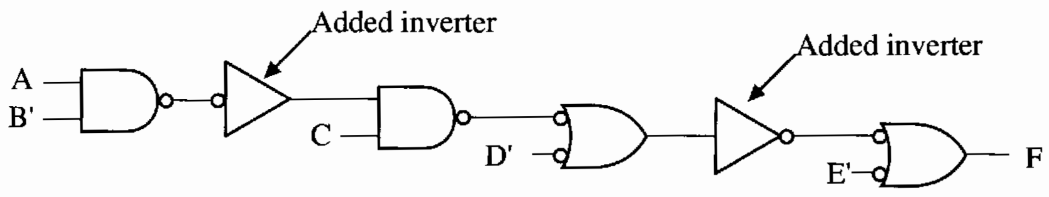
Figure 1-8 Conversion of AND-OR Network to NAND Gates



(a) AND-OR network



(b) First step in NAND conversion



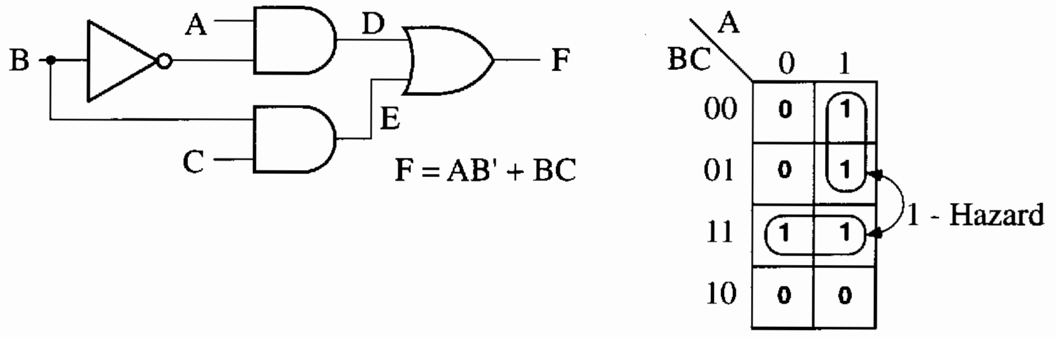
(c) Completed conversion

1.5 HAZARDS IN COMBINATIONAL NETWORKS

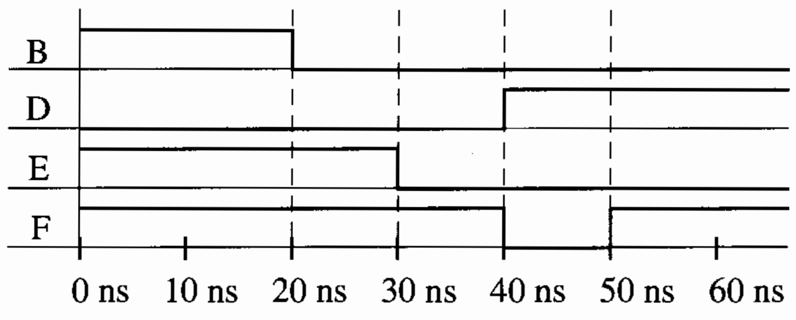
When the input to a combinational network changes, unwanted switching transients may appear in the output. These transients occur when different paths from input to output have different propagation delays. If, in response to an input change and for some combination of propagation delays, a network output may momentarily go to 0 when it should remain a constant 1, we say that the network has a static *1-hazard.* Similarly, if the output may momentarily go to 1 when it should remain a 0, we say that the network has a static *0-hazard.* If, when the output is supposed to change from 0 to 1 (or 1 to 0), the output may change three or more times, we say that the network has a *dynamic hazard.*

Figure 1-9(a) illustrates a network with a static 1-hazard. If $A = C = 1$, the output should remain a constant 1 when B changes from 1 to 0. However, as shown in Figure 1-9(b), if each gate has a propagation delay of 10 ns, E will go to 0 before D goes to 1, resulting in a momentary 0 (a 1-hazard appearing in the output F). As seen on the Karnaugh map, there is no loop that covers both minterm ABC and $AB'C$. So if $A = C = 1$ and B changes, both terms can momentarily go to 0, resulting in a glitch in F. If we add a loop to the map and add the corresponding gate to the network (Figure 1-9(c)), this eliminates the hazard. The term AC remains 1 while B is changing, so no glitch can appear in the output.

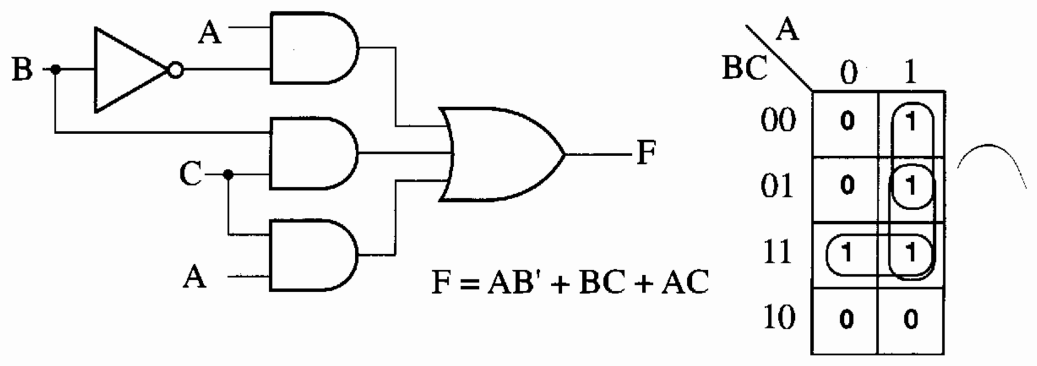
Figure 1-9 Elimination of 1-Hazard



(a) Network with 1-hazard



(b) Timing chart



(c) Network with hazard removed

To design a network that is free of static and dynamic hazards, the following procedure may be used:

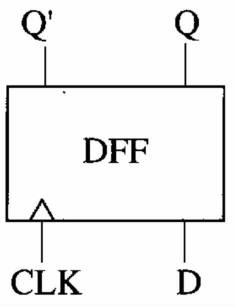
1. Find a sum-of-products expression (F^t) for the output in which every pair of adjacent 1s is covered by a 1-term. (The sum of all prime implicants will always satisfy this condition.) A two-level AND-OR network based on this F^t will be free of 1-, 0-, and dynamic hazards.
2. If a different form of network is desired, manipulate F^t to the desired form by simple factoring, DeMorgan's laws, etc. Treat each x_i and x_i' as independent variables to prevent introduction of hazards.

Alternatively, you can start with a product-of-sums expression in which every pair of adjacent 0s is covered by a 0-term.

1.6 FLIP-FLOPS AND LATCHES

Sequential networks commonly use flip-flops as storage devices. Figure 1-10 shows a clocked D flip-flop. This flip-flop can change state in response to the rising edge of the clock input. The next state of the flip-flop after the rising edge of the clock is equal to the D input before the rising edge. The characteristic equation of the flip-flop is therefore $Q^+ = D$, where Q^+ represents the next state of the Q output after the active edge of the clock and D is the input before the active edge.

Figure 1-10 Clocked D Flip-flop with Rising-edge Trigger

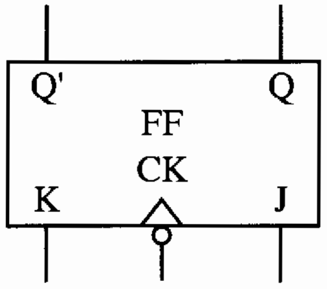


D	Q	Q^+
0	0	0
0	1	0
1	0	1
1	1	1

Figure 1-11 shows a clocked J-K flip-flop and its truth table. Since there is a bubble at the clock input, all state changes occur following the falling edge of the clock input. If $J = K = 0$, no state change occurs. If $J = 1$ and $K = 0$, the flip-flop is set to 1, independent of the present state. If $J = 0$ and $K = 1$, the flip-flop is always reset to 0. If $J = K = 1$, the flip-flop changes state. The characteristic equation, derived from the truth table in Figure 1-11, using a Karnaugh map is

$$Q^+ = JQ' + K'Q. \tag{1-33}$$

Figure 1-11 Clocked J-K Flip-flop



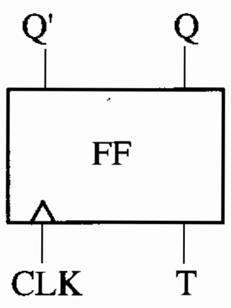
J	K	Q	Q+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

A clocked T flip-flop (Figure 1-12) changes state following the active edge of the clock if $T = 1$, and no state change occurs if $T = 0$. T flip-flops are particularly useful for designing counters. The characteristic equation for the T flip-flop is

$$Q^+ = QT' + Q'T = Q \oplus T \tag{1-34}$$

A J-K flip-flop is easily converted to a T flip-flop by connecting T to both J and K. Substituting T for J and K in (1-33) yields (1-34).

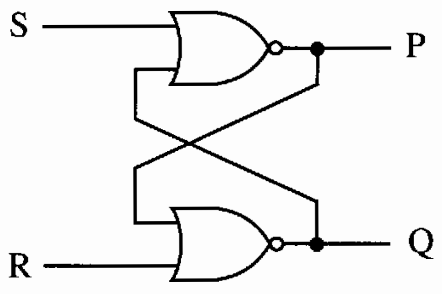
Figure 1-12 Clocked T Flip-flop



T	Q	Q+
0	0	0
0	1	1
1	0	1
1	1	0

Two NOR gates can be connected to form an unclocked S-R (set-reset) flip-flop, as shown in Figure 1-13. An unclocked flip-flop of this type is often referred to as an S-R latch. If $S = 1$ and $R = 0$, the Q output becomes 1 and $P = Q'$. If $S = 0$ and $R = 1$, Q becomes 0 and $P = Q'$. If $S = R = 0$, no change of state occurs. If $R = S = 1$, $P = Q = 0$, which is not a proper flip-flop state, since the two outputs should always be complements. If $R = S = 1$ and these inputs are simultaneously changed to 0, oscillation may occur. For this reason, S and R are not allowed to be 1 at the same time. For purposes of deriving the characteristic equation, we assume the $S = R = 1$ never occurs, in which case $Q^+ = S + R'Q$. In this case, Q^+ represents the state after any input changes have propagated to the Q output.

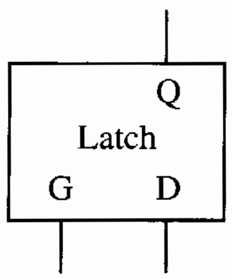
Figure 1-13 S-R Latch



S	R	Q	Q+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	–
1	1	1	–

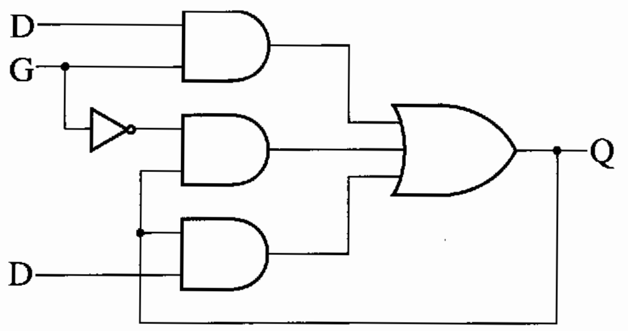
A gated D latch (Figure 1-14), also called a transparent D latch, behaves as follows: If $G = 1$, then the Q output follows the D input ($Q^+ = D$). If $G = 0$, then the latch holds the previous value of Q ($Q^+ = Q$). The characteristic equation for the D latch is $Q^+ = GD + G'Q$. Figure 1-15 shows an implementation of the D latch using gates. Since the Q^+ equation has a 1-hazard, an extra AND gate has been added to eliminate the hazard.

Figure 1-14 Transparent D Latch



G	D	Q	Q+
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 1-15 Implementation of D Latch

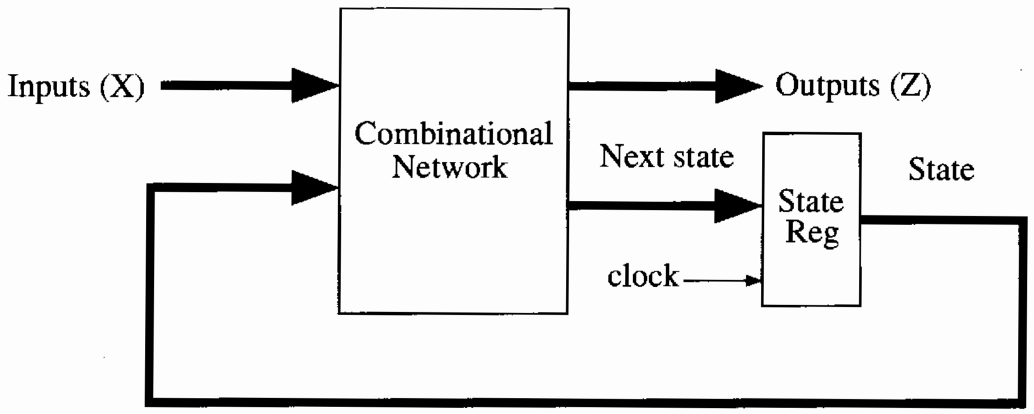


$Q^{+} = DG + G'Q + (DQ)$

1.7 MEALY SEQUENTIAL NETWORK DESIGN

The two basic types of sequential networks are Mealy and Moore. In a Mealy network, the outputs depend on both the present state and the present inputs. In a Moore network, the outputs depend only on the present state. A general model of a Mealy sequential network consists of a combinational network, which generates the outputs and the next state, and a state register, which holds the present state (see Figure 1-16). The state register normally consists of D flip-flops. The normal sequence of events is (1) the *X* inputs are changed to a new value, (2) after a delay, the corresponding *Z* outputs and next state appear at the output of the combinational network, and (3) the next state is clocked into the state register and the state changes. The new state feeds back into the combinational network, and the process is repeated.

Figure 1-16 General Model of Mealy Sequential Machine



As an example of a Mealy sequential network, we will design a code converter that converts an 8-4-2-1 binary-coded-decimal (BCD) digit to an excess-3-coded decimal digit. The input (X) and output (Z) will be serial with the least significant bit first. Table 1-2 lists the desired inputs and outputs at times t_0, t_1, t_2, and t_3. After receiving four inputs, the network should reset to its initial state, ready to receive another BCD digit.

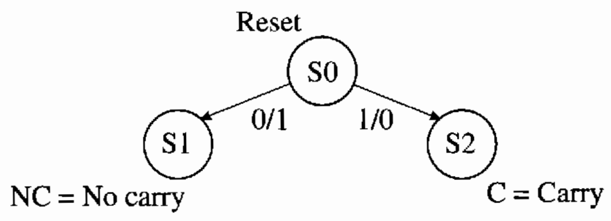
Table 1-2 Code Converter

***X* Input (BCD)**				***Z* Output (excess -3)**			
t_3	t_2	t_1	t_0	t_3	t_2	t_1	t_0
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0

We now construct a state graph for the code converter (Figure 1-17(a)). The excess-3 code is formed by adding 0011 to the BCD digit. For example,

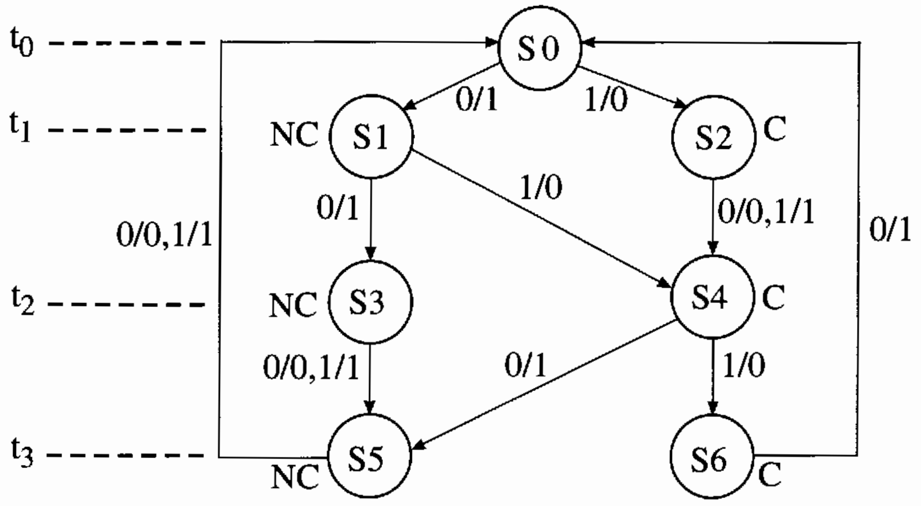
```
 0100        0101
+0011       +0011
 0111        1000
```

At t_0, we add 1 to the least significant bit, so if $X = 0$, $Z = 1$ (no carry), and if $X = 1$, $Z = 0$ (carry = 1). This leads to the following partial state graph:



S0 is the reset state, S1 indicates no carry after the first addition, and S2 indicates a carry of 1. At t_1, we add 1 to the next bit, so if there is no carry from the first addition (state S1), $X = 0$ gives $Z = 0 + 1 + 0 = 1$ and no carry (state S3), and $X = 1$ gives $Z = 1 + 1 + 0 = 0$ and a carry (state S4). If there is a carry from the first addition (state S2), then $X = 0$ gives $Z = 0 + 1 + 1 = 0$ and a carry (S4), and $X = 1$ gives $Z = 1 + 1 + 1 = 1$ and a carry (S4). At t_2, 0 is added to X, and transitions to S5 (no carry) and S6 are determined in a similar manner. At t_3, 0 is again added to X, and the network resets to S0.

Figure 1-17 State Graph and Table for Code Converter



(a) Mealy state graph

	NS		Z	
PS	X = 0	X = 1	X = 0	X = 1
S0	S1	S2	1	0
S1	S3	S4	1	0
S2	S4	S4	0	1
S3	S5	S5	0	1
S4	S5	S6	1	0
S5	S0	S0	0	1
S6	S0	–	1	–

(b) State table

Figure 1-17(b) gives the corresponding state table. (*Fundamentals of Logic Design,* pp. 429–430 gives an alternative way of deriving this state table.) At this point, we should verify that the table has a minimum number of states before proceeding (see Section 1-9). Since the state table has seven states, three flip-flops will be required to realize the table. The next step is to make a state assignment that relates the flip-flop states to the states in the table. The best state assignment to use depends on a number of factors. In many cases, we should try to find an assignment that will reduce the amount of required logic. For some types of programmable logic, a straight binary state assignment will work just as well as any other. For programmable gate arrays, a one-hot assignment (see Section 6.4) may be preferred.

In order to reduce the amount of logic required, we will make a state assignment using the following guidelines (see *Fundamentals of Logic Design,* p. 412):

I. States that have the same next state (NS) for a given input should be given adjacent assignments (look at the columns of the state table).
II. States that are the next states of the same state should be given adjacent assignments (look at the rows).
III. States that have the same output for a given input should be given adjacent assignments.

Using these guidelines tends to clump 1s together on the Karnaugh maps for the next state and output functions. The guidelines indicate that the following states should be given adjacent assignments:

I.	(1, 2), (3, 4), (5, 6)	(in the $X = 1$ column, S_1 and S_2 both have NS S_4; in the $X = 0$ column, S_3 and S_4 have NS S_5, and S_5 and S_6 have NS S_0)
II.	(1, 2), (3, 4), (5, 6)	(S_1 and S_2 are NS of S_0; S_3 and S_4 are NS of S_1; and S_5 and S_6 are NS of S_4)
III.	(0, 1, 4, 6), (2, 3, 5)	

Figure 1-18(a) gives an assignment map, which satisfies the guidelines, and the corresponding transition table. Since state 001 is not used, the next state and outputs for this state are don't cares. The next state and output equations are derived from this table in Figure 1-19. Figure 1-20 shows the realization of the code converter using NAND gates and D flip-flops.

Figure 1-18

$Q_2Q_3 \backslash Q_1$	0	1
00	S0	S1
01		S2
11	S5	S3
10	S6	S4

(a) Assignment map

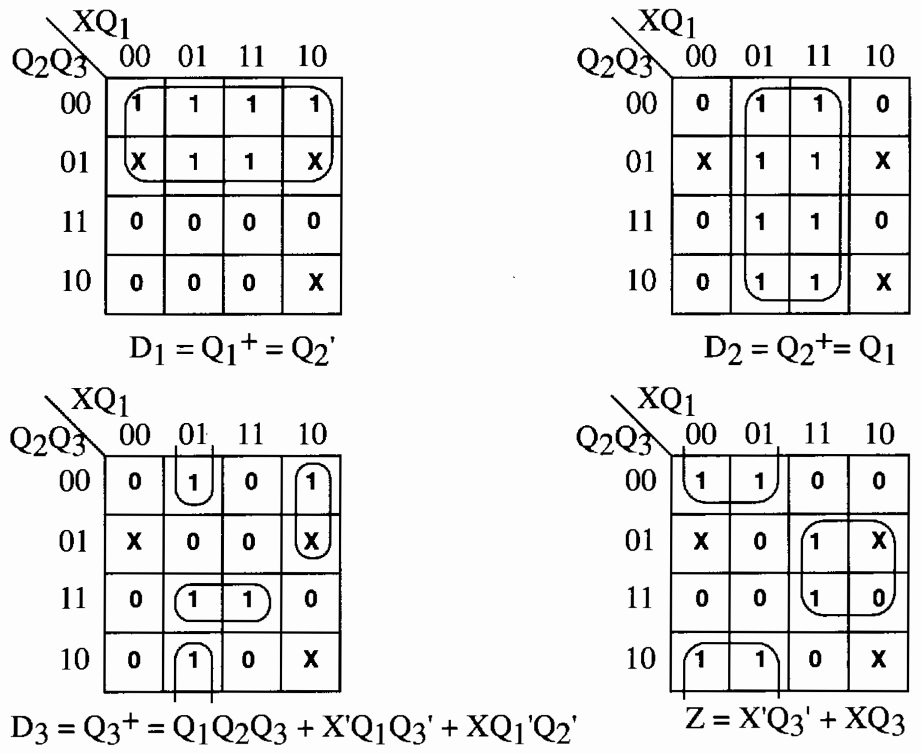
	$Q_1^+Q_2^+Q_3^+$		Z	
$Q_1Q_2Q_3$	X = 0	X = 1	X = 0	X = 1
000	100	101	1	0
100	111	110	1	0
101	110	110	0	1
111	011	011	0	1
110	011	010	1	0
011	000	000	0	1
010	000	xxx	1	x
001	xxx	xxx	x	x

(b) Transition table

If J-K flip-flops are used instead of D flip-flops, the input equations for the J-K flip-flops can be derived from the next state maps. Given the present state flip-flop (Q) and the desired next state (Q^+), the J and K inputs can be determined from the following table, which was derived from the truth table in Figure 1-11:

Q	Q^+	J	K	
0	0	0	X	(No change in Q; J must be 0, K may be 1 to reset Q to 0.)
0	1	1	X	(Change to $Q = 1$; J must be 1 to set or toggle.)
1	0	X	1	(Change to $Q = 0$; K must be 1 to reset or toggle.)
1	1	X	0	(No change in Q; K must be 0, J may be 1 to set Q to 1.)

Figure 1-19 Karnaugh Maps for Figure 1-17



$D_1 = Q_1^+ = Q_2'$

$D_2 = Q_2^+ = Q_1$

$D_3 = Q_3^+ = Q_1Q_2Q_3 + X'Q_1Q_3' + XQ_1'Q_2'$

$Z = X'Q_3' + XQ_3$

Figure 1-20 Realization of Code Converter

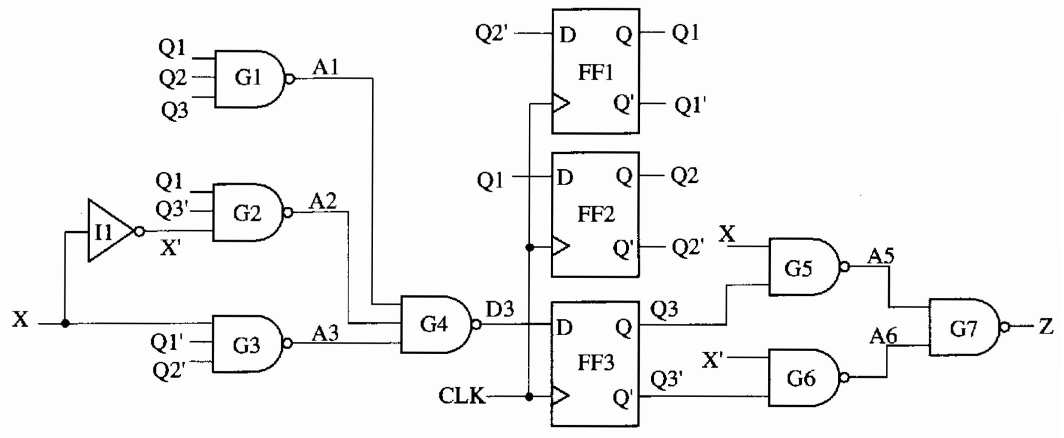
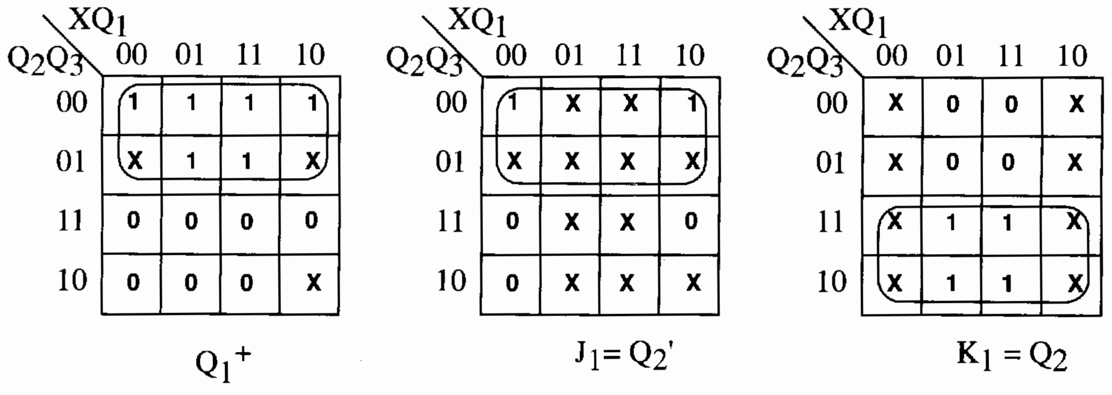
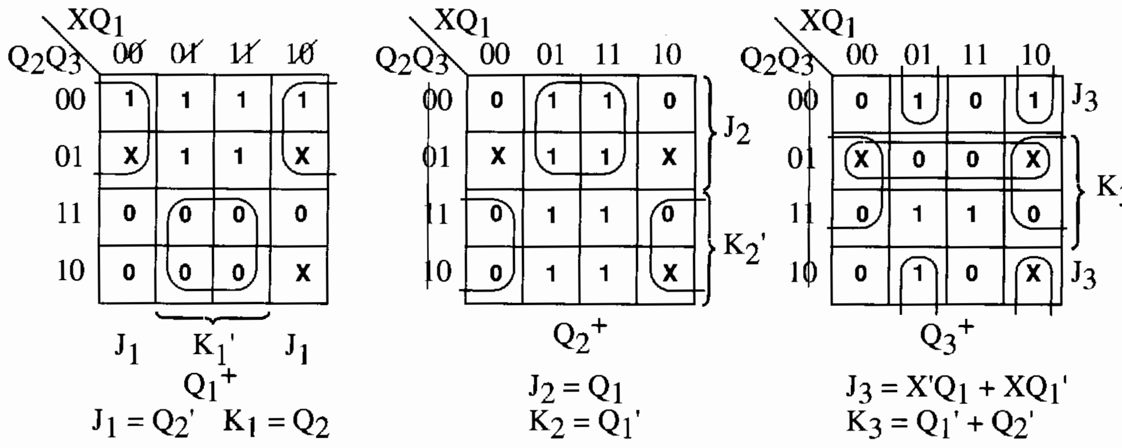


Figure 1-21 shows derivation of J-K flip-flops for the state table of Figure 1-17 using the state assignment of Figure 1-18. First, we derive the J-K input equations for flip-flop Q_1 using the Q_1^+ map as the starting point. From the preceding table, whenever Q_1 is 0, $J = Q_1^+$ and $K = X$. So, we can fill in the $Q_1 = 0$ half of the J_1 map the same as Q_1^+ and the $Q_1 = 0$ half of the K_1 map as all Xs. When Q_1 is 1, $J_1 = X$ and $K_1 = (Q_1^+)'$. So, we can fill in the $Q_1 = 1$ half of the J_1 map with Xs and the $Q_1 = 1$ half of the K_1 map with the complement of the Q_1^+. Since half of every J and K map is don't cares, we can avoid drawing separate J and K maps and read the Js and Ks directly from the Q^+ maps, as illustrated in Figure 1-21(b). This shortcut method is based on the following: If $Q = 0$, then $J = Q^+$, so loop the 1s on the $Q = 0$ half of the map to get J. If $Q = 1$, then $K = (Q^+)'$, so loop the 0s on the $Q = 1$ half of the map to get K. The J and K equations will be independent of Q, since Q is set to a constant value (0 or 1) when reading J and K. To make reading the Js and Ks off the map easier, we cross off the Q values on each map. In effect, using the shortcut method is equivalent to splitting the four-variable Q^+ map into two three-variable maps, one for $Q = 0$ and one for $Q = 1$.

Figure 1-21 Derivation of J-K Input Equations



(a) Derivation using separate J-K maps



(b) Derivation using the shortcut method

The following summarizes the steps required to design a sequential network:

1. Given the design specifications, determine the required relationship between the input and output sequences. Then find a state graph and state table.
2. Reduce the table to a minimum number of states. First eliminate duplicate rows by row matching; then form an implication table and follow the procedure in Section 1.9.
3. If the reduced table has m states ($2^{n-1} < m \leq 2^n$), n flip-flops are required. Assign a unique combination of flip-flop states to correspond to each state in the reduced table.
4. Form the transition table by substituting the assigned flip-flop states for each state in the reduced state tables. The resulting transition table specifies the next states of the flip-flops and the output in terms of the present states of the flip-flops and the input.
5. Plot next-state maps and input maps for each flip-flop and derive the flip-flop input equations. Derive the output functions.
6. Realize the flip-flop input equations and the output equations using the available logic gates.
7. Check your design using computer simulation or another method.

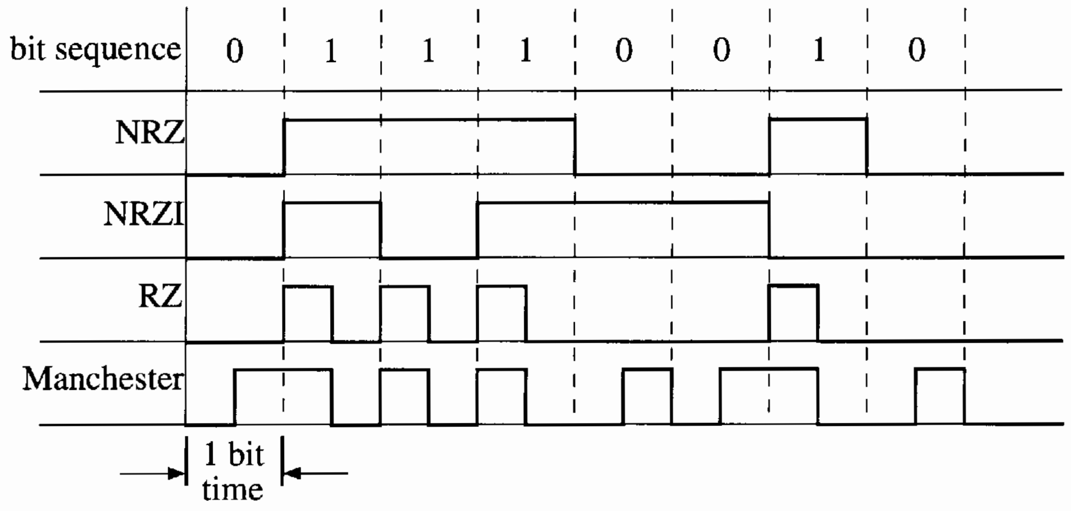
Steps 2 through 7 may be carried out using a suitable CAD program.

1.8 DESIGN OF A MOORE SEQUENTIAL NETWORK

As an example of designing a Moore sequential machine, we will design a converter for serial data. Binary data is frequently transmitted between computers as a serial stream of bits. Figure 1-22 shows four different coding schemes for serial data. The example shows transmission of the bit sequence 0, 1, 1, 1, 0, 0, 1, 0. With the NRZ (nonreturn-to-zero) code, each bit is transmitted for one bit time without any change. With the NRZI (nonreturn-to-zero-inverted) code, data is encoded by the presence or absence of transitions in the data signal. For each 0 in the original sequence, the bit transmitted is the same as the previous bit transmitted. For each 1 in the original sequence, the bit transmitted is the complement of the previous bit transmitted. For the RZ (return-to-zero) code, a 0 is transmitted as 0 for one full bit time, but a 1 is transmitted as a 1 for the first half of the bit time, and then the signal returns to 0 for the second half. For the Manchester code, a 0 is transmitted as 0 for the first half of the bit time and a 1 for the second half, but a 1 is transmitted as a 1 for the first half and a 0 for the second half. Thus, the Manchester encoded bit always changes in the middle of the bit time.

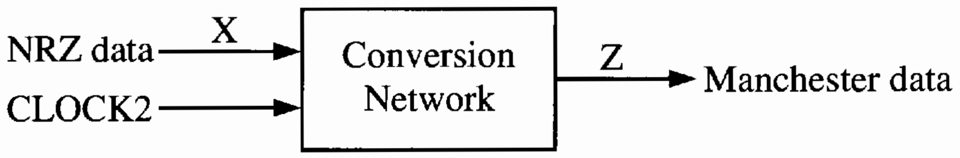
We will design a Moore sequential network that converts an NRZ-coded bit stream to a Manchester-coded bit stream (Figure 1-23). In order to do this, we will use a clock (*CLOCK2*) that is twice the frequency of the basic bit clock. If the NRZ bit is 0, it will be 0 for two *CLOCK2* periods, and if it is 1, it will be 1 for two *CLOCK2* periods. Thus, starting in the reset state (S_0), the only two possible input sequences are 00 and 11, and the corresponding output sequences are 01 and 10. When a 0 is received, the network goes to S_1 and outputs a 0; when the second 0 is received, it goes to S_2 and outputs a 1. Starting in S_0, if a 1 is received, the network goes to S_3 and outputs a 1, and when the second 1 is received, it must go to a state with a 0 output. Going back to S_0 is appropriate since S_0 has

Figure 1-22 Coding Schemes for Serial Data Transmission

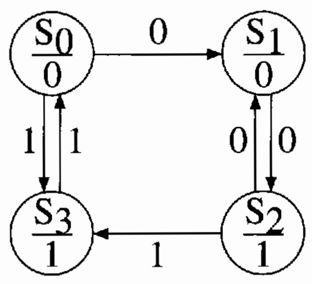


a 0 output and the network is ready to receive another 00 or 11 sequence. When in S_2, if a 00 sequence is received, the network can go to S_1 and back to S_2. If a 11 sequence is received in S_2, the network can go to S_3 and then back to S_0. The corresponding Moore state table has two don't cares, which correspond to input sequences that cannot occur.

Figure 1-23 Moore network for NRZ-to-Manchester Conversion



(a) Conversion network



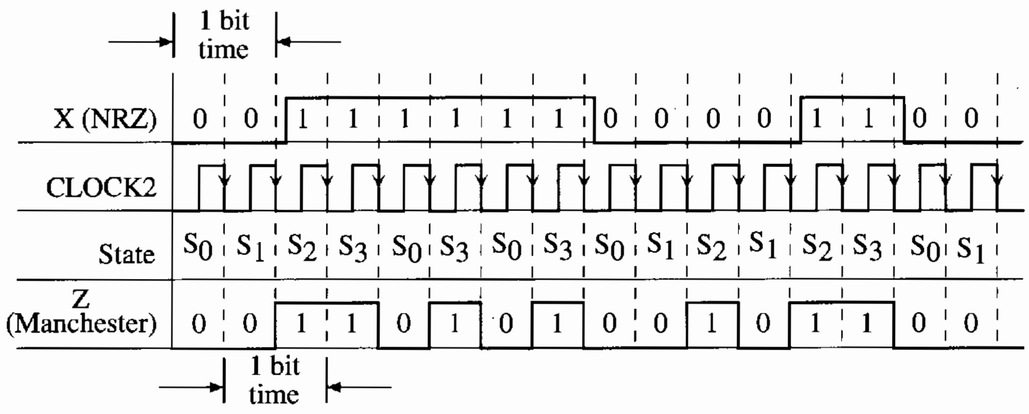
(b) State graph

Present State	Next State X = 0	Next State X = 1	Present Output (Z)
S_0	S_1	S_3	0
S_1	S_2	–	0
S_2	S_1	S_3	1
S_3	–	S_0	1

(c) State table

Figure 1-24 shows the timing chart for the Moore network. Note that the Manchester output is shifted one clock time with respect to the NRZ input. This shift occurs because a Moore network cannot respond to an input until the active edge of the clock occurs. This is in contrast to a Mealy network, for which the output can change after the input changes and before the next clock.

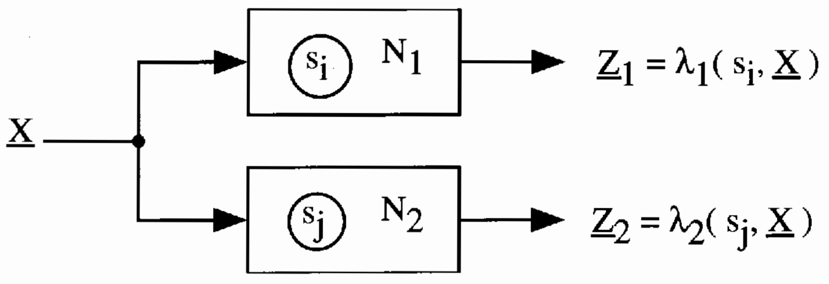
Figure 1-24 Timing for Moore Network



1.9 EQUIVALENT STATES AND REDUCTION OF STATE TABLES

The concept of equivalent states is important for the design and testing of sequential networks. Two states in a sequential network are said to be *equivalent* if we cannot tell them apart by observing input and output sequences. Consider two sequential networks, N_1 and N_2 (see Figure 1-25). N_1 and N_2 could be copies of the same network. N_1 is started in state s_i, and N_2 is started in state s_j. We apply the same input sequence, $\underline{X}$, to both networks and observe the output sequences, $\underline{Z}_1$ and $\underline{Z}_2$. (The underscore notation indicates a sequence.) If $\underline{Z}_1$ and $\underline{Z}_2$ are the same, we reset the networks to states s_i and s_j, apply a different input sequence, and observe $\underline{Z}_1$ and $\underline{Z}_2$. If the output sequences are the same for all possible input sequences, we say the s_i and s_j are equivalent ($s_i \equiv s_j$). Formally, we can define equivalent states as follows: $s_i \equiv s_j$ if and only if, for every input sequence $\underline{X}$, the output sequences $\underline{Z}_1 = \lambda_1(s_i, \underline{X})$ and $\underline{Z}_2 = \lambda_2(s_j, \underline{X})$ are the same. This is not a very practical way to test for state equivalence since, at least in theory, it requires input sequences of infinite length. In practice, if we have a bound on number of states, then we can limit the length of the test sequences.

Figure 1-25 Sequential Networks


$\underline{X}$
s_i
N_1
$\underline{Z}_1 = \lambda_1(s_i, \underline{X})$
s_j
N_2
$\underline{Z}_2 = \lambda_2(s_j, \underline{X})$

A more practical way to determine state equivalence uses the state equivalence theorem: $s_i \equiv s_j$ if and only if for every single input X, the outputs are the same and the next states are equivalent. When using the definition of equivalence, we must consider all input sequences, but we do not need any information about the internal state of the system. When using the state equivalence theorem, we must look at both the output and next state, but we need to consider only single inputs rather than input sequences.

The table of Figure 1-26(a) can be reduced by eliminating equivalent states. First, observe that states a and h have the same next states and outputs when $X = 0$ and also when $X = 1$. Therefore, a ≡ h so we can eliminate row h and replace h with a in the table. To determine if any of the remaining states are equivalent, we will use the state equivalence theorem. From the table, since the outputs for states a and b are the same, a ≡ b if and only if c ≡ d and e ≡ f. We say that c–d and e–f are implied pairs for a–b. To keep track of the implied pairs, we make an implication chart, as shown in Figure 1-26(b). We place c–d and e–f in the square at the intersection of row a and column b to indicate the implication. Since states d and e have different outputs, we place an × in the d–e square to indicate that d ≢ e. After completing the implication chart in this way, we make another pass through the chart. The e–g square contains c–e and b–g. Since the c–e square has an ×, c ≢ e, which implies e ≢ g, so we × out the e–g square. Similarly, since e ≢ f, we × out the f–g square. On the next pass through the chart, we × out all the squares that contain e–f or f–g as implied pairs (shown on the chart with dashed ×s). In the next pass, no additional squares are ×ed out, so the process terminates. Since all the squares corresponding to non-equivalent states have been ×ed out, the coordinates of the remaining squares indicate equivalent state pairs. From the first column, a ≡ b; from third column, c ≡ d; and from the fifth column, e ≡ f.

The implication table method of determining state equivalence can be summarized as follows:

1. Construct a chart that contains a square for each pair of states.
2. Compare each pair of rows in the state table. If the outputs associated with states i and j are different, place an × in square i–j to indicate that i ≢ j. If the outputs are the same, place the implied pairs in square i–j. (If the next states of i and j are m and n for some input x, then m–n is an implied pair.) If the outputs and next states are the same (or if i–j implies only itself), place a check (√) in square i–j to indicate that i ≡ j.
3. Go through the table square by square. If square i–j contains the implied pair m–n, and square m–n contains an ×, then i ≢ j, and an × should be placed in square i–j.
4. If any ×s were added in step 3, repeat step 3 until no more ×s are added.
5. For each square i–j that does not contain an ×, i ≡ j.

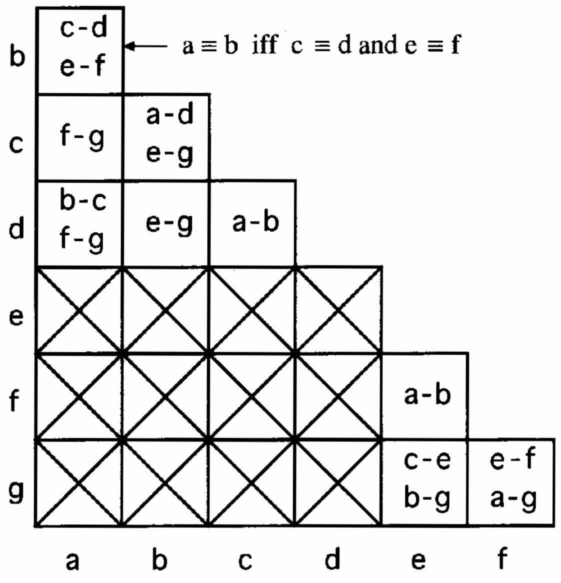
If desired, row matching can be used to partially reduce the state table before constructing the implication table. Although we have illustrated this procedure for a Mealy table, the same procedure applies to a Moore table.

Two sequential networks are said to be equivalent if every state in the first network has an equivalent state in the second network, and vice versa.

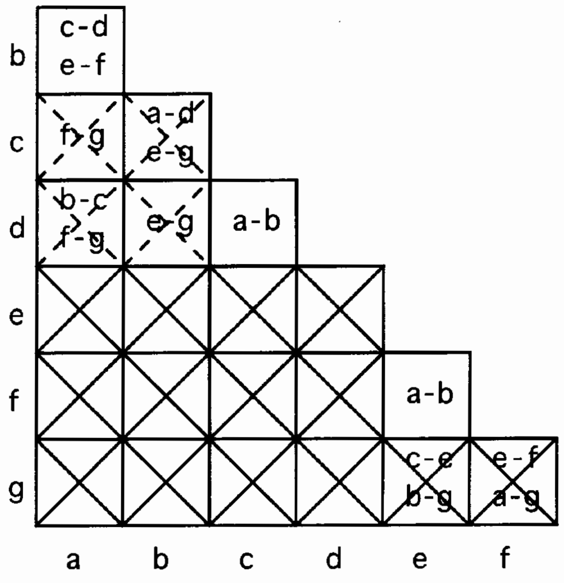
Figure 1-26 State Table Reduction

Present State	Next State $X = 0$	Next State 1	Present Output $X = 0$	Present Output 1
a	c	f	0	0
b	d	e	0	0
c	~~h~~a	g	0	0
d	b	g	0	0
e	e	b	0	1
f	f	a	0	1
g	c	g	0	1
~~h~~	~~c~~	~~f~~	~~0~~	~~0~~

(a) State table reduction by row matching



(b) Implication chart (first pass)



(c) After second and third passes

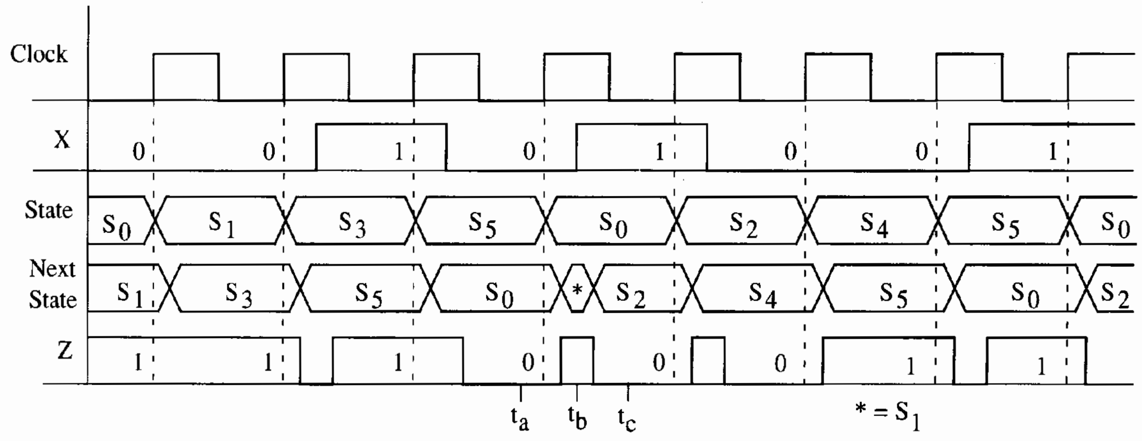
	$X =$ 0	1	$X =$ 0	1
a	c	e	0	0
c	a	g	0	0
e	e	a	0	1
g	c	g	0	1

(d) Final reduced table

1.10 SEQUENTIAL NETWORK TIMING

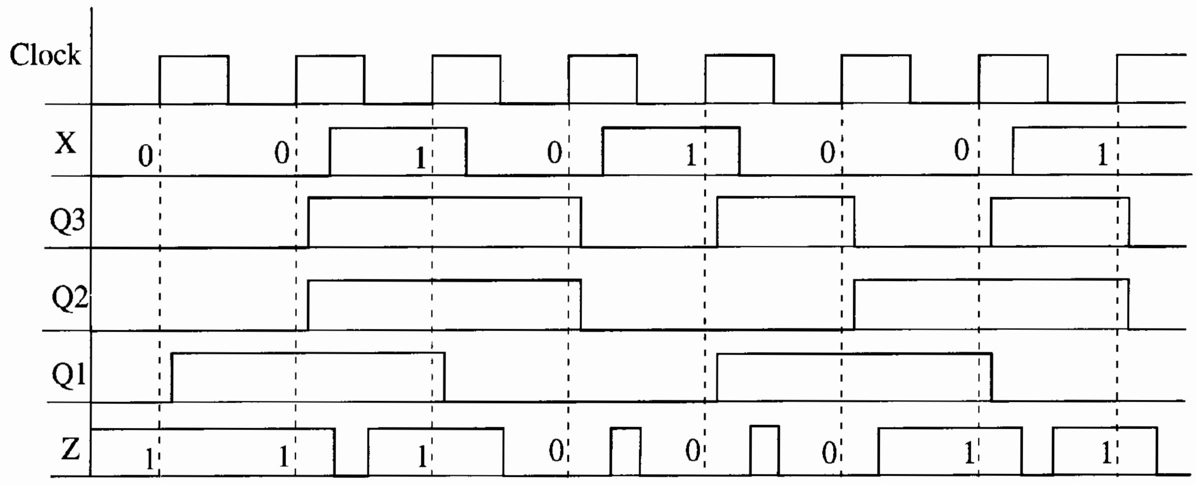
If the state table of Figure 1-17(b) is implemented in the form of Figure 1-16, the timing waveforms are as shown in Figure 1-27. Propagation delays in the network have been neglected. In this example, the input sequence is 00101001, and *X* is assumed to change in the middle of the clock pulse. At any given time, the next state and *Z* output can be read from the next state table. For example, at time t_a, State = S_5 and $X = 0$, so Next State = S_0 and $Z = 0$. At time t_b following the rising edge of the clock, State = S_0 and *X* is still 0, so Next State = S_1 and $Z = 1$. Then *X* changes to 1, and at time t_c Next State = S_2 and $Z = 0$. Note that there is a *glitch* (sometimes called a false output) at t_b. The *Z* output momentarily has an incorrect value at t_b, because the change in *X* is not exactly synchronized with the active edge of the clock. The correct output sequence, as indicated on the waveform, is 1 1 1 0 0 0 1 1. Several glitches appear between the correct outputs; however, these are of no consequence if *Z* is read at the right time. The glitch in the next state at t_b (S_1) also does not cause a problem, because the next state has the correct value at the active edge of the clock.

Figure 1-27 Timing Diagram for Code Converter



The timing waveforms derived from the network of Figure 1-20 are shown in Figure 1-28. They are similar to the general timing waveforms given in Figure 1-27 except that State has been replaced with the states of the three flip-flops, and a propagation delay of 10 ns has been assumed for each gate and flip-flop.

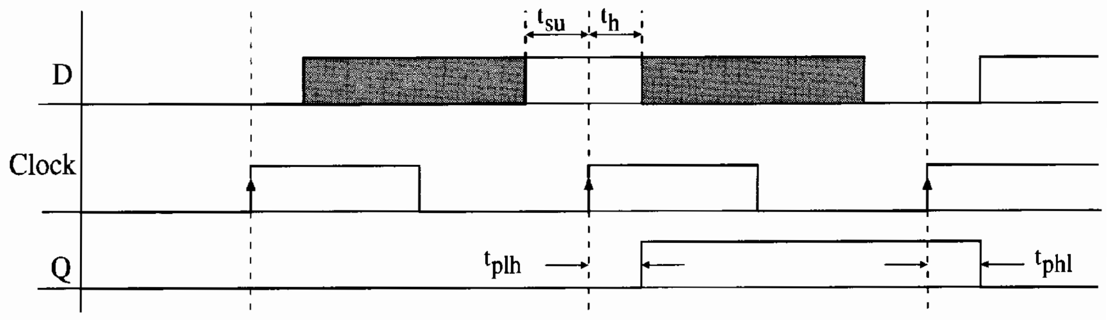
Figure 1-28 Timing Diagram for Figure 1-20



1.11 SETUP AND HOLD TIMES

For an ideal D flip-flop, if the *D* input changed at exactly the same time as the active edge of the clock, the flip-flop would operate correctly. However, for a real flip-flop, the *D* input must be stable for a certain amount of time before the active edge of the clock (called the setup time). Furthermore, *D* must be stable for a certain amount of time after the active edge of the clock (called the hold time). Figure 1-29 illustrates setup and hold times for a *D* flip-flop that changes state on the rising edge of the clock. *D* can change at any time during the shaded region on the diagram, but it must be stable during the time interval t_{su} before the active edge and for t_h after the active edge. If *D* changes at any time during the forbidden interval, it cannot be determined whether the flip-flop will change state. Even worse, the flip-flop may malfunction and output a short pulse or even go into oscillation.

Figure 1-29 Setup and Hold Times for D Flip-flop



The propagation delay from the time the clock changes to the time the *Q* output changes is also indicated in Figure 1-29. The propagation delay for a low-to-high change in *Q* is t_{plh}, and for a high-to-low change it is t_{phl}. Minimum values for t_{su} and t_h and maximum values for t_{plh} and t_{phl} can be read from manufacturers' data sheets.

The maximum clock frequency for a sequential network depends on several factors. For a network of the form of Figure 1-16, assume that the maximum propagation delay through the combinational network is $t_{c\max}$ and the maximum propagation delay from the time the clock changes to the flip-flop output changes is $t_{p\max}$, where $t_{p\max}$ is the maximum of t_{plh} and t_{phl}. Then the maximum time from the active edge of the clock to the time the change in Q propagates back to the D flip-flop inputs is $t_{p\max} + t_{c\max}$. If the clock period is t_{ck}, the D inputs must be stable t_{su} before the end of the clock period. Therefore,

$$t_{p\max} + t_{c\max} \le t_{ck} - t_{su}$$

and

$$t_{ck} \ge t_{p\max} + t_{c\max} + t_{su}$$

For example, for the network of Figure 1-20, if the maximum gate delay is 15 ns, $t_{p\max}$ for the flip-flops is 15 ns, and t_{su} is 5 ns, then

$$t_{ck} \ge 2 \times 15 + 15 + 5 = 50 \text{ ns.}$$

The maximum clock frequency is then $1/t_{ck} = 20$ MHz. Note that the inverter is not in the feedback loop.

A hold-time violation could occur if the change in Q fed back through the combinational network and caused D to change too soon after the clock edge. The hold time is satisfied if

$$t_{p\min} + t_{c\min} \ge t_h$$

When checking for hold-time violations, the worst case occurs when the timing parameters have their minimum values. Since $t_{p\min} > t_h$ for normal flip-flops, a hold-time violation due to Q changing does not occur. However, a setup or hold-time violation could occur if the X input to the network changes too close to the active edge of the clock.

When the X input to a sequential network changes, we must make sure that the input change propagates to the flip-flop inputs such that the setup time is satisfied before the active edge of the clock. If X changes at time t_x before the active edge of the clock (see Figure 1-30), then the setup time is satisfied if

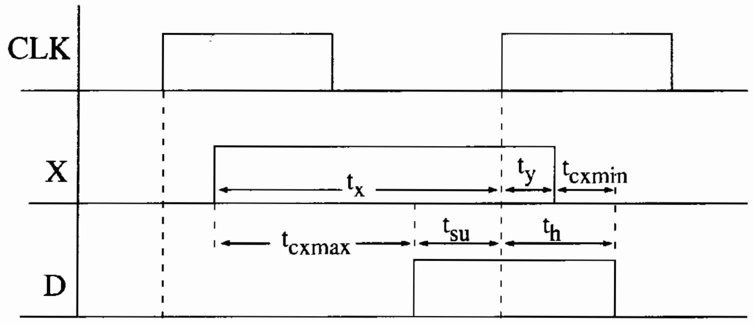
$$t_x \ge t_{cx\max} + t_{su}$$

where $t_{cx\max}$ is the maximum propagation delay from X to the flip-flop input. In order to satisfy the hold time, we must make sure that X does not change too soon after the clock. If X changes at time t_y after the active edge of the clock, then the hold time is satisfied if

$$t_y \ge t_h - t_{cx\min}$$

where $t_{cx\min}$ is the minimum propagation delay from X to the flip-flop input. If t_y is negative, X can change before the active clock edge and still satisfy the hold time.

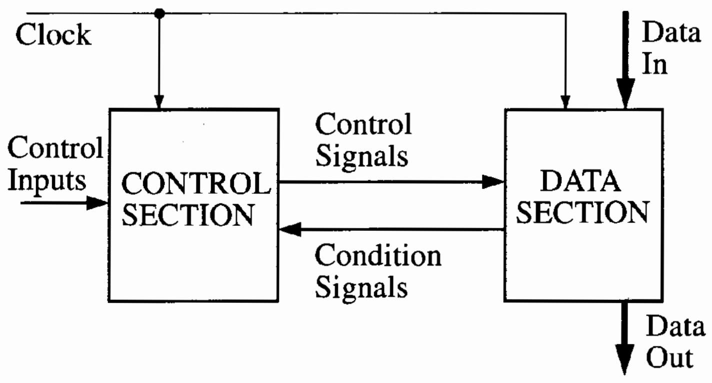
Figure 1-30 Setup and Hold Timing for Changes in *X*



1.12 SYNCHRONOUS DESIGN

One of the most commonly used digital design techniques is *synchronous design*. This type of design uses a clock to synchronize the operation of all flip-flops, registers, and counters in the system. All state changes will occur immediately following the active edge of the clock. The clock period must be long enough so that all flip-flop and register inputs will have time to stabilize before the next active edge of the clock.

Figure 1-31 Synchronous Digital System



A typical digital system can be divided into a control section and a data section, as shown in Figure 1-31. A common clock synchronizes the operation of the control and data sections. The data section may contain data registers, arithmetic units, counters, etc. The control section is a sequential machine that generates control signals to control the operation of the data section. For example, if the data section contains a shift register, the control section may generate signals *Ld* and *Sh*, which determine when the register is to be loaded and when it is to be shifted. The data section may generate condition signals that effect the control sequence. For example, if a data operation produces an arithmetic overflow, then the data section might generate a condition signal *V* to indicate an overflow.

Figure 1-32 Timing Chart for System with Falling-Edge Devices

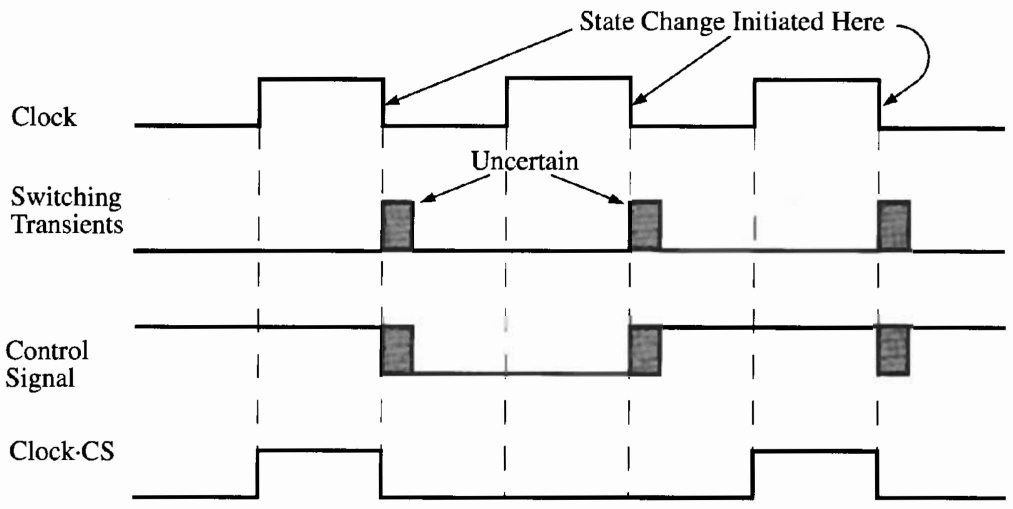
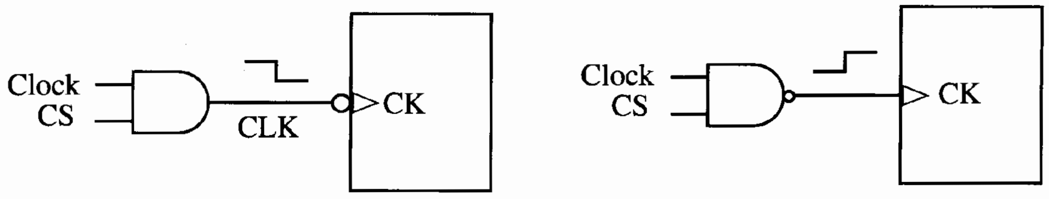


Figure 1-32 illustrates the operation of a digital system, which uses devices that change state on the falling edge of the clock. Several flip-flops may change state in response to this falling edge. The time at which each flip-flop changes state is determined by the propagation delay for that flip-flop. The changes in flip-flop states in the control section will propagate through the combinational network that generates the control signals, and some of the control signals may change as a result. The exact times at which the control signals change depend on the propagation delays in the gate networks that generate the signals as well as the flip-flop delays. Thus, after the falling edge of the clock, there is a period of uncertainty during which control signals may change. Glitches and spikes may occur in the control signals due to hazards. Furthermore, when signals are changing in one part of the circuit, noise may be induced in another part of the circuit. As indicated by the cross-hatching in Figure 1-32, there is a time interval after each falling edge of the clock in which there may be noise in a control signal (*CS*), and the exact time at which the control signal changes is not known.

If we want a device in the data section to change state on the falling edge of the clock only if the control signal *CS* = 1, we can AND the clock with *CS*, as shown in Figure 1-33(a). The *CLK* input to the device will be a clean signal, and except for a small delay in the AND gate, the transitions will occur in synchronization with the clock. The *CLK* signal is clean because the clock is 0 during the time interval in which the switching transients occur in *CS*.

Figure 1-33 Gated Control Signal



(a) Falling-edge device **(b)** Rising-edge device

Figure 1-34 illustrates the operation of a digital system that uses devices that change state on the rising edge of the clock. In this case, the switching transients that result in noise and uncertainty will occur following the rising edge of the clock. The cross-hatching indicates the time interval in which the control signal *CS* may be noisy. If we want a device to change state on the rising edge of the clock when $CS = 1$, it is tempting to AND the clock with *CS*, as shown in Figure 1-35. The resulting signal, which goes to the *CK* input of the device, may be noisy and timed incorrectly. In particular, the *CLK1* pulse at (a) will be short and noisy. It may be too short to trigger the device, or it may be noisy and trigger the device more than once. In general, it will be out of synch with the clock, because the control signal does not change until after some of the flip-flops in the control network have changed state. The rising edge of the pulse at (b) again will be out of synch with the clock, and it may be noisy. But even worse, the device will trigger near point (b) when it should not trigger there at all. Since $CS = 0$ at the time of the rising edge of the clock, triggering should not occur until the next rising edge, when $CS = 1$.

Figure 1-34 Timing Chart for System with Rising-Edge Devices

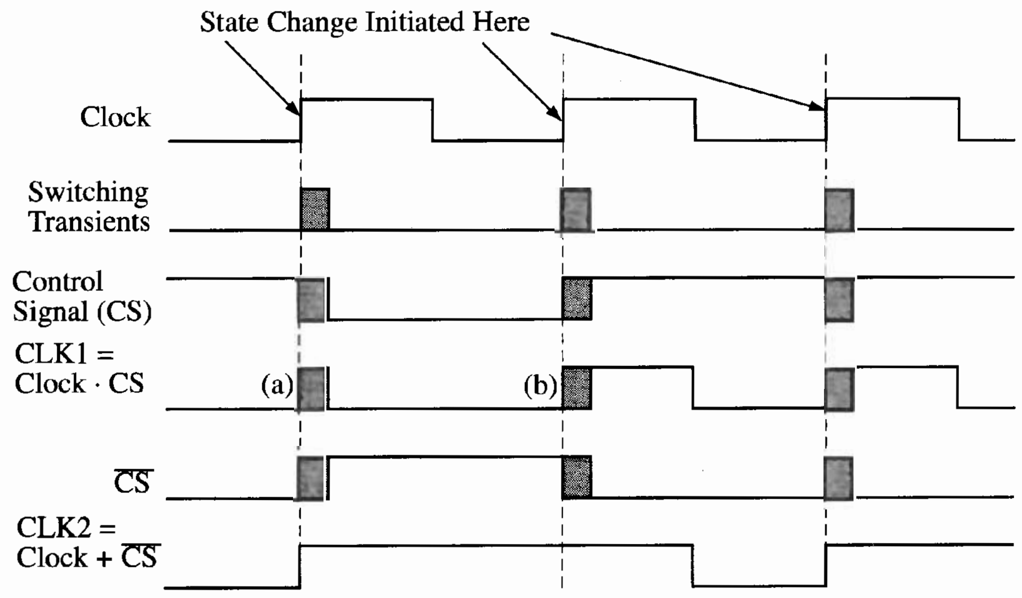
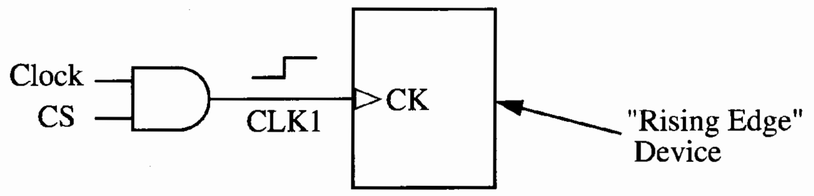


Figure 1-35 Incorrect Design for Rising-Edge Device

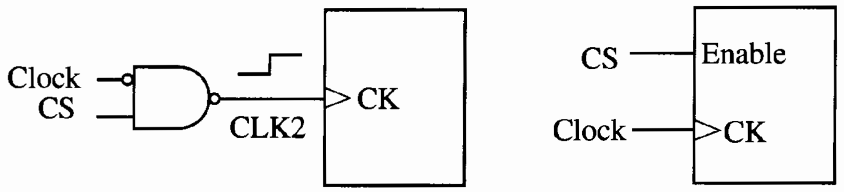


If we move the bubble in Figure 1-33(a) from the device input to the gate output (see Figure 1-33(b)), the timing will be the same as in Figure 1-32. We now have a rising-edge device, but it will trigger on the falling edge of the system clock. To get around this problem, we can invert the clock, as indicated by the added bubble in Figure 1-36(a). The *CK* input to the device is then

$$CLK2 = (CS \cdot clock')' = CS' + clock$$

As shown in Figure 1-34, the *CLK2* signal will be free of noise, and when $CS = 1$, *CLK2* will change from 0 to 1 at the same time as the clock.

Figure 1-36 Correct Design for Rising-Edge Device



(a) With gated clock **(b)** With enable

Many registers, counters, and other devices used in synchronous systems have an enable input (see Figure 1-36(b)). When enable = 1, the device changes state in response to the clock, and when enable = 0, no state change occurs. Use of the enable input eliminates the need for a gate on the clock input, and the associated timing problems are avoided.

In summary, synchronous design is based on the following principles:

• Method:	All clock inputs to flip-flops, registers, counters, etc., are driven directly from the system clock or from the clock ANDed with a control signal.
• Result:	All state changes occur immediately following the active edge of the clock signal.
• Advantage:	All switching transients, switching noise, etc., occur between clock pulses and have no effect on system performance.

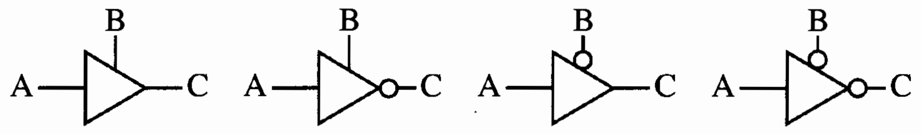
Asynchronous design (see *Fundamentals of Logic Design,* Chapters 23–27) is generally more difficult than synchronous design. Since there is no clock to synchronize the state changes, problems may arise when several state variables must change at the same time. A race occurs if the final state depends on the order in which the variables change. Asynchronous design requires special techniques to eliminate problems with races and hazards. On the other hand, synchronous design has several disadvantages: In high-speed circuits where the propagation delay in the wiring is significant, the clock signal must be carefully routed so that it reaches all the clock inputs at essentially the same time. The maximum clock rate is determined by the worst-case delay of the longest path. The system inputs may not be synchronized with the clock, so use of synchronizers may be required.

1.13 TRISTATE LOGIC AND BUSSES

In digital systems, transferring data back and forth between several system components is often necessary. In this section, we introduce the concept of tristate buffers and show how tristate busses can be used to facilitate data transfers between registers.

Figure 1-37 shows four kinds of tristate buffers. *B* is a control input used to enable or disable the buffer output. When a buffer is enabled, the output (*C*) is equal to the input (*A*) or its complement. When a buffer is disabled, the output is in a high-impedance, or hi-Z, state, which is equivalent to an open circuit. Normally, if we connect the outputs of two gates or flip-flops together, the circuit will not operate properly. However, we can connect two tristate buffer outputs, provided that only one output is enabled at a time.

Figure 1-37 Four Kinds of Tristate Buffers



B	A	C
0	0	Hi-Z
0	1	Hi-Z
1	0	0
1	1	1

(a)

B	A	C
0	0	Hi-Z
0	1	Hi-Z
1	0	1
1	1	0

(b)

B	A	C
0	0	0
0	1	1
1	0	Hi-Z
1	1	Hi-Z

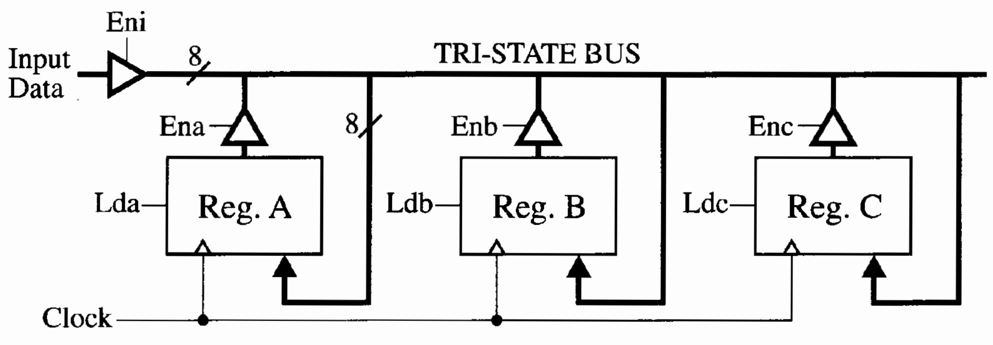
(c)

B	A	C
0	0	1
0	1	0
1	0	Hi-Z
1	1	Hi-Z

(d)

Figure 1-38 shows a system with three registers connected to a tristate bus. Each register is 8 bits wide, and the bus consists of 8 wires connected in parallel. Each tristate buffer symbol in the figure represents 8 buffers operating in parallel with a common enable input. Only one group of buffers is enabled at a time. For example, if *Enb* = 1, the register *B* output is driven onto the bus. The data on the bus is routed to the inputs of register *A*, register *B*, and register *C*. However, data is loaded into a register only when its load input is 1 and the register is clocked. Thus, if *Enb* = *Ldc* = 1, the data in register *B* will be copied into register *C* when the active edge of the clock occurs. If *Eni* = *Lda* = *Ldb* = 1, the input data will be loaded in registers *A* and *B* when the registers are clocked.

Figure 1-38 Data Transfer Using Tristate Bus



Problems

1.1 Write out the truth table for the following equation.

$$F = (A \oplus B) \cdot C + A' \cdot (B' \oplus C)$$

1.2 A full subtracter computes the difference of three inputs X, Y, and B_{in}, where $Diff = X - Y - B_{in}$. When $X < (Y + B_{in})$, the borrow output B_{out} is set. Fill in the truth table for the subtracter and derive the sum-of-products and product-of-sums equations for *Diff* and B_{out}.

1.3 Simplify Z using a 4-variable map with map-entered variables. $ABCD$ represents the state of a control network. Assume that the network can never be in state 0100, 0001, or 1001.

$$Z = BC'DE + ACDF' + ABCD'F' + ABC'D'G + B'CD + ABC'D'H'$$

1.4 For the following functions, find the minimum sum of products using 4-variable maps with map-entered variables. In (a) and (b), m_i represents a minterm of variables A, B, C, and D.

(a) $F(A, B, C, D, E) = \Sigma m(0, 4, 6, 13, 14) + \Sigma d(2, 9) + E(m_1 + m_{12})$

(b) $Z(A, B, C, D, E, F, G) = \Sigma m(2, 5, 6, 9) + \Sigma d(1, 3, 4, 13, 14) + E(m_{11} + m_{12}) + F(m_{10}) + G(m_0)$

(c) $H = A'B'CDF' + A'CD + A'B'CD'E + BCDF'$

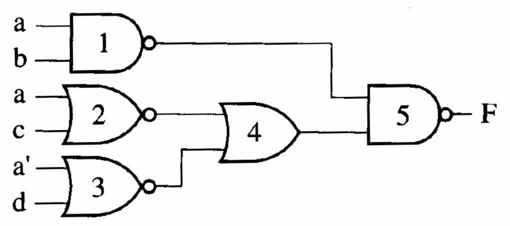
(d) $G = C'E'F + DEF + AD'E'F' + BC'E'F + AD'EF'$

Hint: Which variables should be used for the map sides and which variables should be entered into the map?

1.5

(a) Find all the static hazards in the following network. For each hazard, specify the values of the input variables and which variable is changing when the hazard occurs. For one of the hazards, specify the order in which the gate outputs must change.

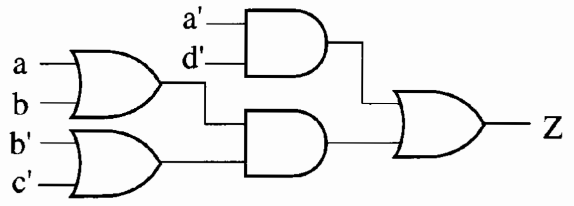
(b) Design a NAND-gate network that is free of static hazards to realize the same function.



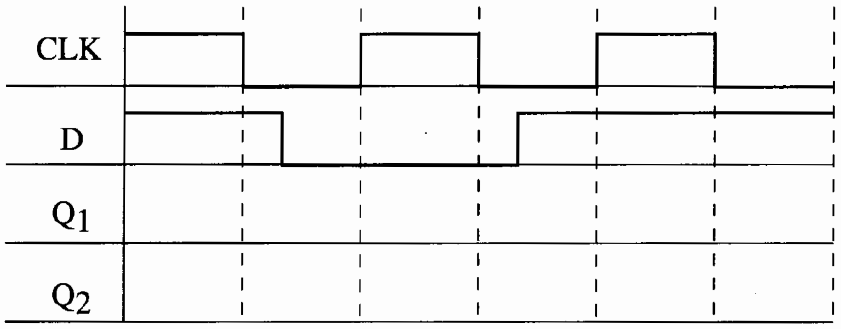
1.6

(a) Find all the static hazards in the following network. State the condition under which each hazard can occur.

(b) Redesign the network so that it is free of static hazards. Use gates with at most three inputs.



1.7 Construct a clocked D flip-flop, triggered on the rising edge of *CLK*, using two transparent *D* latches and any necessary gates. Complete the following timing diagram, where Q_1 and Q_2 are latch outputs. Verify that the flip-flop output changes to *D* after the rising edge of the clock.



1.8 A synchronous sequential network has one input and one output. If the input sequence 0101 or 0110 occurs, an output of two successive 1s will occur. The first of these 1s should occur coincident with the last input of the 0101 or 0110 sequence. The network should reset when the second 1 output occurs. For example,

```
input sequence:    X =  010011101010 101101 ...
output sequence:   Z =  000000000011 000011 ...
```

(a) Derive a Mealy state graph and table with a minimum number of states (6 states).

(b) Try to choose a good state assignment. Realize the network using J-K flip-flops and NAND gates. Repeat using NOR gates. (Work this part by hand.)

(c) Check your answer to (b) using the *LogicAid* program. Also use the program to find the NAND solution for two other state assignments.

1.9 A sequential network has one input (X) and two outputs (Z_1 and Z_2). An output $Z_1 = 1$ occurs every time the input sequence 010 is completed provided that the sequence 100 has never occurred. An output $Z_2 = 1$ occurs every time the input sequence 100 is completed. Note that once a $Z_2 = 1$ output has occurred, $Z_1 = 1$ can never occur, but *not* vice versa.

(a) Derive a Mealy state graph and table with a minimum number of states (8 states).

(b) Try to choose a good state assignment. Realize the network using J-K flip-flops and NAND gates. Repeat using NOR gates. (Work this part by hand.)

(c) Check your answer to (b) using the *LogicAid* program. Also use the program to find the NAND solution for two other state assignments.

1.10 A sequential network has one input (X) and two outputs (S and V). X represents a 4-bit binary number N, which is input least significant bit first. S represents a 4-bit binary number equal to $N + 2$, which is output least significant bit first. At the time the fourth input occurs, $V = 1$ if $N + 2$ is too large to be represented by 4 bits; otherwise, $V = 0$. The value of S should be the proper value, not a don't care, in both cases. The network always resets after the fourth bit of X is received.

(a) Derive a Mealy state graph and table with a minimum number of states (6 states).

(b) Try to choose a good state assignment. Realize the network using J-K flip-flops and NAND gates. Repeat using NOR gates. (Work this part by hand.)

(c) Check your answer to (b) using the *LogicAid* program. Also use the program to find the NAND solution for two other state assignments.

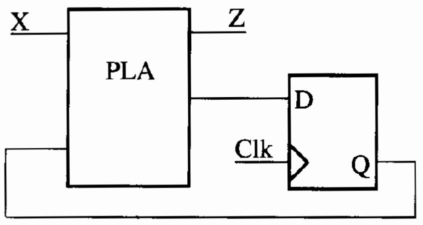
1.11 A sequential network has one input (X) and two outputs (D and B). X represents a 4-bit binary number N, which is input least significant bit first. D represents a 4-bit binary number equal to $N - 2$, which is output least significant bit first. At the time the fourth input occurs, $B = 1$ if $N - 2$ is negative; otherwise, $B = 0$. The network always resets after the fourth bit of X is received.

(a) Derive a Mealy state graph and table with a minimum number of states (6 states).

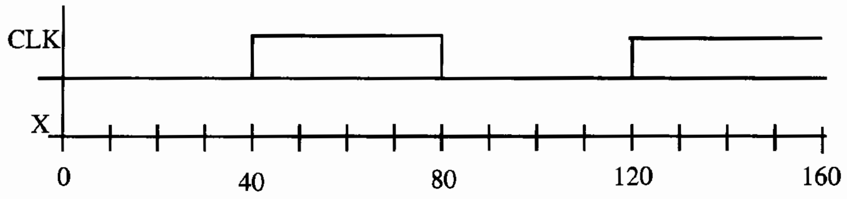
(b) Try to choose a good state assignment. Realize the network using J-K flip-flops and NAND gates. Repeat using NOR gates. (Work this part by hand.)

(c) Check your answer to (b) using the *LogicAid* program. Also use the program to find the NAND solution for two other state assignments.

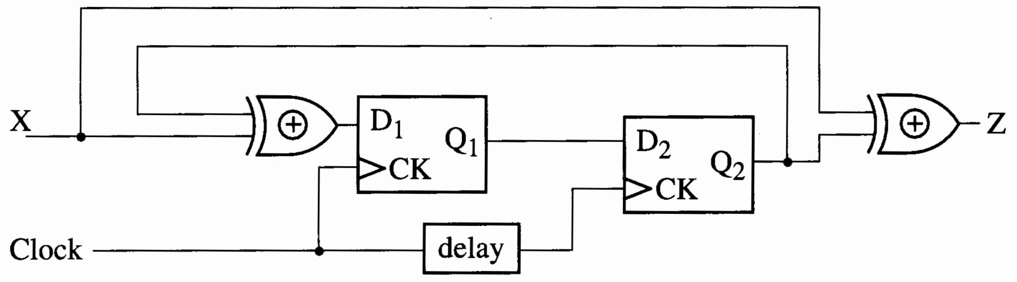
1.12 A sequential network has the following form. The delay through the combinational network is in the range $5 \le t_c \le 20$ ns. The propagation delay from the rising edge of the clock to the change in the flip-flop output is in the range $5 \le t_p \le 10$ ns. The required setup and hold times for the flip-flop are $t_{su} = 10$ ns and $t_h = 5$ ns. Indicate on the diagram the times at which X is allowed to change.



Q	0	1	0	1
0	0	1	0	1
1	0	0	1	0
	Q^+		Z	



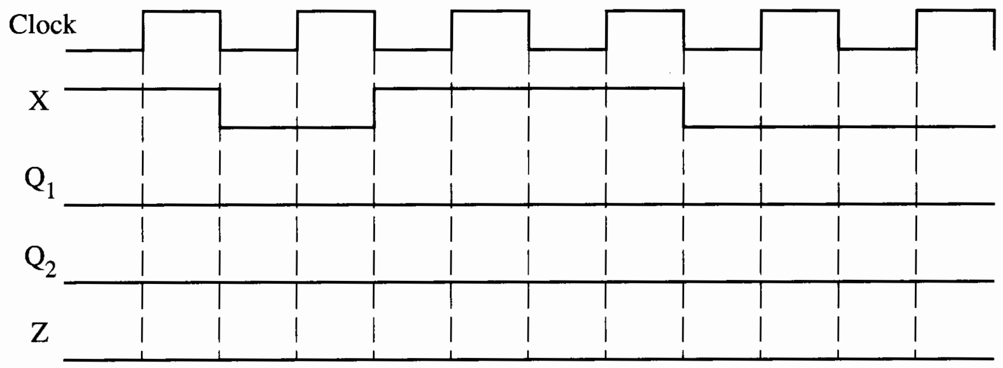
1.13 A Mealy sequential network is implemented using the network shown below. Assume that if the input X changes, it changes at the same time as the **falling** edge of the clock. Assume the following delays: XOR gate, 5 to 15 ns; flip-flop propagation delay, 5 to 15 ns; setup time, 10 ns; hold time, 5 ns. Initially assume that the "delay" is 0 ns.



(a) Determine the maximum clock rate for proper synchronous operation.

(b) Assume a clock period of 100 ns. What is the maximum value that "delay" can have and still achieve proper synchronous operation?

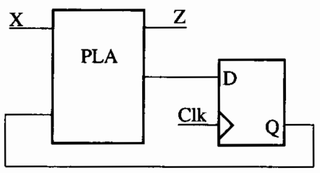
(c) Complete the following timing diagram. Indicate the proper times to read the output (Z). Assume that "delay" is 0 ns and that the propagation delay for the flip-flop and XOR gate has a nominal value of 10 ns.



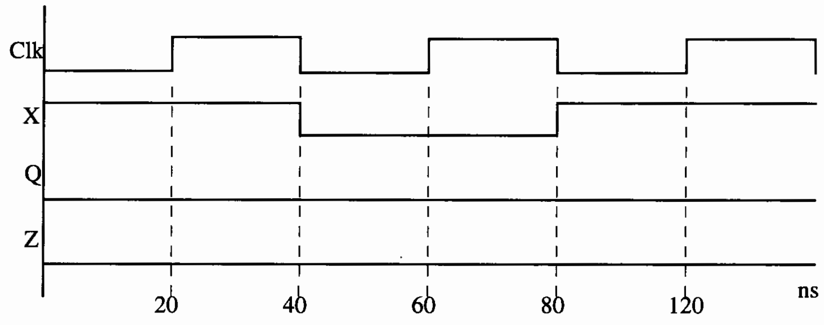
1.14 A sequential network consists of a PLA and a D flip-flop, as shown.

(a) Complete the timing diagram assuming that the propagation delay for the PLA is in the range 5 to 10 ns, and the propagation delay from clock to output of the D flip-flop is 5 to 10 ns. Use cross-hatching on your timing diagram to indicate the intervals in which Q and Z can change, taking the range of propagation delays into account.

(b) Assuming that X always changes at the same time as the falling edge of the clock, what is the maximum setup and hold time specification that the flip-flop can have and still maintain proper operation of the network?



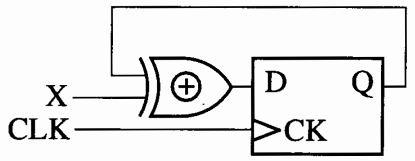
Q	0	1	0	1
0	0	1	0	1
1	0	0	1	0
	Q^+		Z	



1.15 A D flip-flop has a setup time of 4 ns, a hold time of 2 ns, and a propagation delay from the rising edge of the clock to the change in flip-flop output in the range of 6 to 12 ns. The XOR gate delay is in the range of 1 to 8 ns.

(a) What is the minimum clock period for proper operation of the following network?

(b) What is the earliest time after the rising clock edge that X is allowed to change?

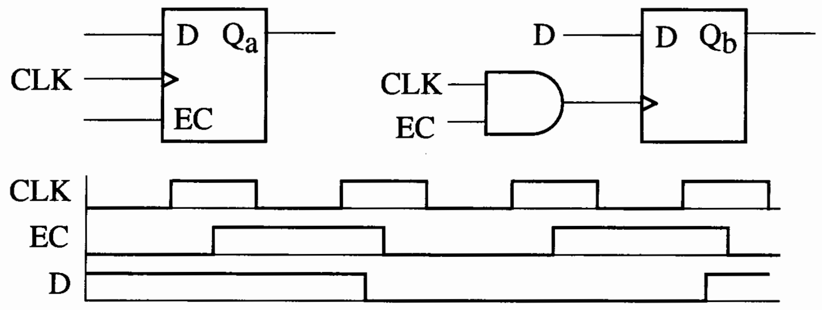


1.16

(a) Do the following two networks have essentially the same timing?

(b) Draw the timing for Q_a and Q_b given the timing diagram.

(c) If your answer to (a) is no, show what change(s) should be made in the second network so that the two networks have essentially the same timing (do not change the flip-flop).



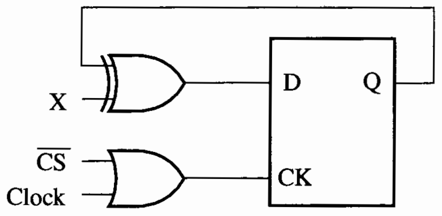
1.17

Assume that CS (and also $\overline{CS}$) change 2 ns after the rising edge of the clock.

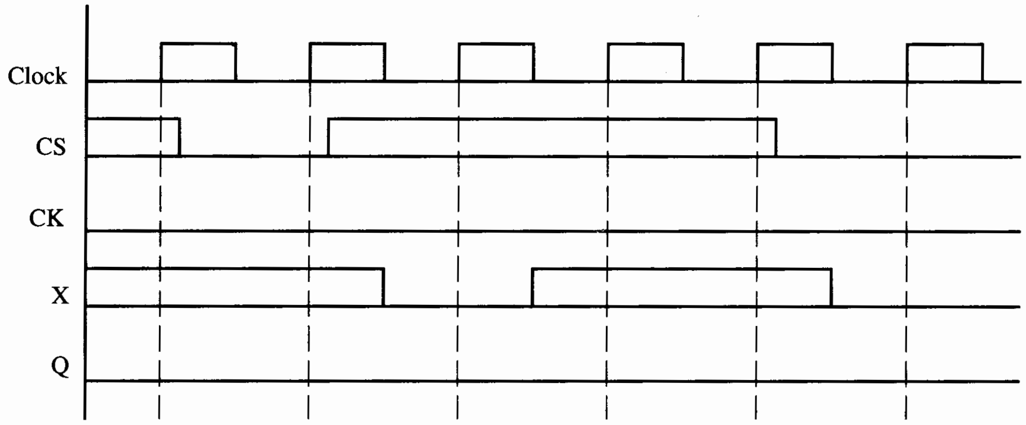
(a) Plot CK and *Q* on the timing diagram. A precise plot is not required; just show the relative times at which the signals change.

(b) If *X* changes at the falling edge of *Clock*, as shown, what is the maximum clock frequency?

(c) With respect to the rising edge of *Clock*, what is the earliest that *X* can change and still satisfy the hold-time requirement?



flip-flop propagation delay = 10 to 15 ns
setup time = 4 ns
hold time = 2 ns
XOR gate delay = 4 to 8 ns
OR gate delay = 2 to 6 ns


Clock
CS
CK
X
Q